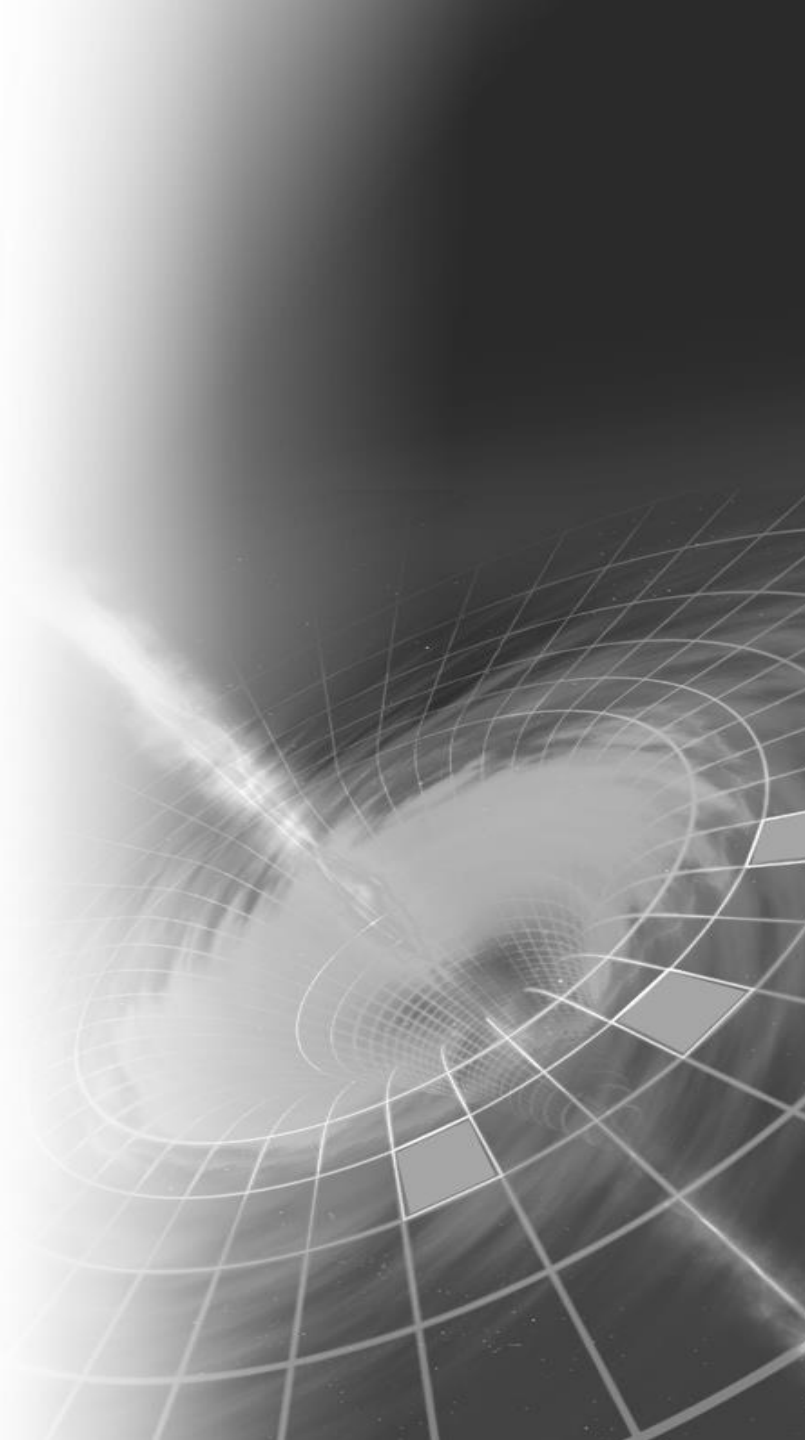
Bachelor-Programm

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 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

Umsetzung mit Bison

Eingabe … **while (C)** S1

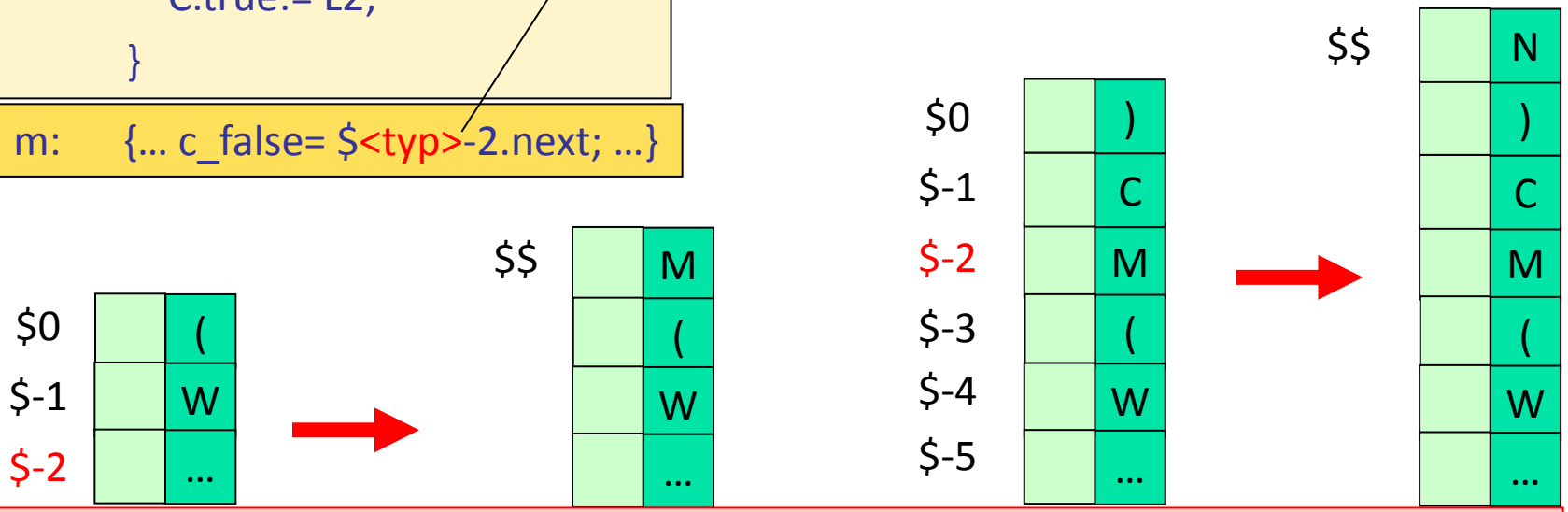
S → WHILE '(' M C ')' N S1

kritischer Fall: $0, $-1, $-2, …

Type-Cast

```
M → ε { L1:= newLabel();
        L2:= newLabel();
        C.false:= val[tos-3].next;
        C.true:= L2;
      }
m:    {… c_false= $<typ>-2.next; …}
```

```
N → ε { S1.next= val[tos-3].L1;
      }
n:    { $$.S1_next= $<typ>-2. L1; }
```



Achtung Fehlerquelle:

(1) allgemeine Kellerindizierung stimmt **nicht** konsequent mit Bison-Indizierung überein !

(2) Zugriff auf Komponenten **außerhalb** der aktuellen Regel verlangt eine explizite Typwandlung (ohne Test)

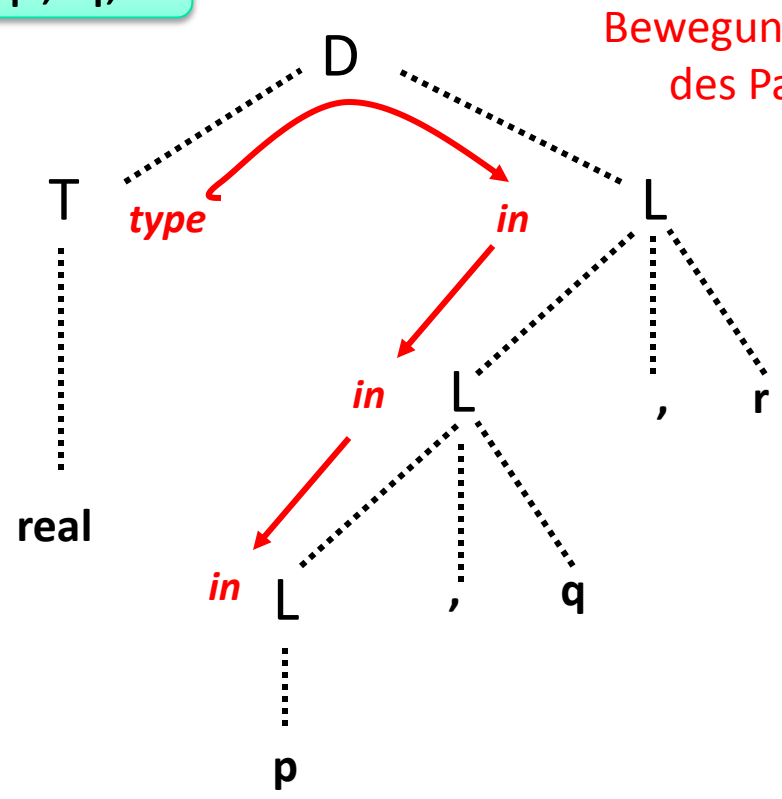
Effizienzsteigerung von Attributberechnungen:

erneut: Variablendeklarations-Beispiel

Eingabe **real** p, q, r

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)

Symboltabellen-Update



Welche Zustände(Symbole)
kommen wann/wie auf den Ableitungsstack?

➔ Kenntnis hilft, um auf zugriffs-synchrone Attributwert zugreifen zu können

Auswertung ererbter Attribute

Eingabe	Zustand	benutzte Produktion, semantische Regel
real p, q, r	-	
p, q, r	**real**	
p, q, r	T	T → **real** T.type:= real
, q, r	T **p**	
, q, r	T L	L → id addtype(id.entry, L.in)
q, r	T L **,**	
, r	T L **, q**	
, r	T L	L → L **,** id L1.in= L.in; addtype(id.entry, L.in)
r	T L **,**	
	T L **, r**	
	T L	L → L **,** id L1.in= L.in; addtype(id.entry, L.in)
	D	D → T L L.in := T.type

?

Auswertung ererbter Attribute

Eingabe	Zustand	benutzte Produktion, semantische Regel
real p, q, r	-	
p, q, r	**real**	
p, q, r	T	T → **real** T.type:= real
, q, r	T **p**	
, q, r	T L	L → id addtype(id.entry, L.in)
q, r	T L **,**	
, r	T L **, q**	
, r	T L	L → L **,** id L1.in= L.in; addtype(id.entry, L.in)
r	T L **,**	
	T L **, r**	
	T L	L → L **,** id L1.in= L.in; addtype(id.entry, L.in)
	D	D → T L L.in := T.type

immer
wenn zu L reduziert wird,
steht T an
einer bestimmten Position

➔ gesicherter Zugriff auf T.type möglich

(also direkter Zugriff!)

Simulation ererbter Attribute: Beispiel

Eingabe	Keller	Produktion
real p, q, r	-	
p, q, r	real	
p, q, r	T	T → real
, q, r	T p	
, q, r	T L	L → id
q, r	T L ,	
, r	T L , q	
, r	T L	L → L , id
r	T L ,	
	T L , r	
	T L	L → L , id
	D	D → T L

Beobachtung

- T.type ist immer auf festem Platz im Keller (bezüglich tos) zu finden
  - bei "L → L , id" ist dies tos - 3
  - bei "L → id" ist dies tos - 1

Folgerung

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

Simulation ererbter Attribute erhöht i.allg. das Laufzeitverhalten eines Compilers

Auswertung ererbter Attri

Eingabe	Zustand	benutzte Produktion, se
real p, q, r	-	
p, q, r	**real**	
p, q, r	T	T → **real** T.type:= re
, q, r	T **p**	
, q, r	T L	L → id addtype(id.entry, L.in)
q, r	T L **,**	
, r	T L **, q**	
, r	T L	L → L **,** id L1.in= L.in; addtype(id.entry, L.in)
r	T L **,**	
	T L **, r**	
	T L	L → L **,** id L1.in= L.in; addtype(id.entry, L.in)
	D	D → T L ~~L.in := T.type~~

Beobachtung

- T.type ist immer auf festem Platz im Keller (bezüglich tos) zu finden
  - bei "L → L , id" ist dies tos - 3
  - bei "L → id" ist dies tos - 1

Folgerung

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

Simulation ererbter Attribute erhöht i.allg. das Laufzeitverhalten eines Compilers

Simulation ererbter Attribute: Beispiel

angepasste semantische Aktionen

Produktion	Semantische Regel - Codefragment	Bison
D → T L		
T → **int**	val[ntos] = integer	$$= integer
T → **real**	val[ntos] = real	$$= real
L → L_1 , **id**	addtype (val[tos], val[tos-3])	addtype ($3, $<typ>-2)
L → **id**	addtype (val[tos], val[tos-1])	addtype ($1, $<typ>0)

tos = momentaner »top-of-stack«

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

Positionen
in der aktuellen Regel
positiv

Positionen
vor der aktuellen Regel
0, negativ

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

Übersetzung arithmetischer Ausdrücke

Idee:

1. dynamische Bereitstellung von temporären Adressen zur Speicherung von Zwischenergebnissen der Verkettung

   Befehl: newTemp() // liefert neue temporäre Adresse

2. Meta-Symbol E

   2 synthetische Attribute: E.code // 3-Adresscode für E

   E.addr // Adresse, für Resultat der Ausführung von E.code

Fazit: S-Attributierte Grammatik

Produktion	Semantische Regel
$S \rightarrow id = E$	S.code= E.code || top.get(id.lexem) || '=' || E.addr
$E \rightarrow E_1 + E_2$	E.addr= newTemp() E.code= E_1.code || E_2.code || E.addr || '=' || E_1.addr || '+' || E_2.addr
$E \rightarrow - E_1$	E.addr= newTemp() E.code= E_1.code || E.addr || '=' '-' || E_1.addr
$E \rightarrow (E_1)$	E.addr= E_1.addr E.code= E_1.code
$E \rightarrow id$	E.addr= top.get(id.lexem) E.code= ""

Übersetzung arithmetischer Ausdrücke

Produktion	Semantische Regel
$S \rightarrow id = E$	S.code= E.code || top.get(id.lexem) || '=' || E.addr
$E \rightarrow E_1 + E_2$	E.addr= **new** Temp() E.code= E_1.code || E_2.code || E.addr || '=' || E_1.addr || '+' || E_2.addr
$E \rightarrow - E_1$	E.addr= **new** Temp() E.code= E_1.code || E.addr || '=' '-' || E_1.addr
$E \rightarrow (E_1)$	E.addr= E_1.addr E.code= E_1.code
$E \rightarrow id$	E.addr= top.get(id.lexem) E.code= ""

Übersetzung arithmetischer Ausdrücke

Produktion	Semantische Regel
$S \rightarrow id = E$	S.code= E.code || top.get(id.lexem) || '=' || E.addr
$E \rightarrow E_1 + E_2$	E.addr= **new** Temp() E.code= E_1.code || E_2.code || E.addr || '=' || E_1.addr || '+' || E_2.addr
$E \rightarrow - E_1$	E.addr= **new** Temp() E.code= E_1.code || E.addr || '=' '-' || E_1.addr
$E \rightarrow (E_1)$	E.addr= E_1.addr E.code= E_1.code
$E \rightarrow id$	E.addr= top.get(id.lexem) E.code= ""

Übersetzung arithmetischer Ausdrücke

Produktion	Semantische Regel
$S \rightarrow id = E$	S.code= E.code || top.get(id.lexem) || '=' || E.addr
$E \rightarrow E_1 + E_2$	E.addr= **new** Temp() E.code= E_1.code || E_2.code || E.addr || '=' || E_1.addr || '+' || E_2.addr
$E \rightarrow - E_1$	E.addr= **new** Temp() E.code= E_1.code || E.addr || '=' '-' || E_1.addr
$E \rightarrow (E_1)$	E.addr= E_1.addr E.code= E_1.code
$E \rightarrow id$	E.addr= top.get(id.lexem) E.code= ""

Übersetzung arithmetischer Ausdrücke

Produktion	Semantische Regel
$S \rightarrow id = E$	S.code= E.code || top.get(id.lexem) || '=' || E.addr
$E \rightarrow E_1 + E_2$	E.addr= **new** Temp() E.code= E_1.code || E_2.code || E.addr || '=' || E_1.addr || '+' || E_2.addr
$E \rightarrow - E_1$	E.addr= **new** Temp() E.code= E_1.code || E.addr || '=' '-' || E_1.addr
$E \rightarrow (E_1)$	E.addr= E_1.addr E.code= E_1.code
$E \rightarrow id$	E.addr= top.get(id.lexem) E.code= ""

Übersetzung arithmetischer Ausdrücke

Produktion	Semantische Regel
$S \rightarrow id = E$	S.code= E.code || top.get(id.lexem) || '=' || E.addr
$E \rightarrow E_1 + E_2$	E.addr= **new** Temp() E.code= E_1.code || E_2.code || E.addr || '=' || E_1.addr || '+' || E_2.addr
$E \rightarrow - E_1$	E.addr= **new** Temp() E.code= E_1.code || E.addr || '=' '-' || E_1.addr
$E \rightarrow (E_1)$	E.addr= E_1.addr E.code= E_1.code
$E \rightarrow id$	E.addr= top.get(id.lexem) E.code= ""

Übersetzung boolscher Ausdrücke

Vorkommen:

1. Berechnung boolscher Werte (erfolgt analog zu arithmetischen Ausdrücken)

2. Steuerung des Kontrollflusses (als Bedingung in **if-then-else**, **while**, …)

   man erzeugt ebenfalls sog. Sprung-Code

Metasymbol B bekommt 3 Attribute

- B.code //synthetisch: 3-Adress-Code für B
- B.l_true // ererbt: Marke, zu der B.code springt, wenn B-Code-Ausführung **true** liefert
- B.l_false // ererbt: Marke, zu der B.code springt, wenn B-Code-Ausführung **false** liefert

Frage: Wo kommen B.l_true und B_l_false her?

- entweder von einem umfassenderen boolschen Ausdruck oder
- von der Anweisung, in der B als boolscher Ausdruck enthalten ist

Anweisung S hat zwei Attribute (code und Sprungmarke)

Sprung-Code für boolsche Ausdrücke

Produktion	Semantische Regel
$B \rightarrow B_1 \,||\, B_2$	B_1.l_true || B.l_true B_1.l_false = **new** Label() B_2.l_true = B.l_true B_2.l_false = B.l_false B.code = B_1.code || B_1.l_false || B_2.code
$B \rightarrow B_1$ && B_2	B_1.l_true= **new** Label() B_1.l_false = B.l_false B_2.l_true = B.l_true B_2.l_false = B.l_false B.code = B_1.code || B_1.l_true || B_2.code
$B \rightarrow$! B_1	B_1.l_true = B.l_false B_1.l_false = B.l_true B.code = B_1.code
$B \rightarrow E_1$ **rel** E_2	B.code = E_1.code || E_2.l_true || *gen* ('if' E1.addr **rel**.op E2.addr 'goto' B.l_true) || *gen* ('goto' B.l_true)
$B \rightarrow$ **true**	B.code= *gen* ('goto' B.l_true)
$B \rightarrow$ **false**	B.code= *gen* ('goto' B.l_false)

Sprung-Code für boolsche Ausdrücke

Produktion	Semantische Regel
$B \rightarrow B_1 \,||\, B_2$	B_1.l_true = B.l_true B_1.l_false = **new** Label() B_2.l_true = B.l_true B_2.l_false = B.l_false B.code = B_1.code || B_1.l_false || B_2.code
$B \rightarrow B_1$ && B_2	B_1.l_true= **n** B_1.l_false = B_2.l_true = B B_2.l_false = B.code = B_1.
$B \rightarrow$! B_1	B_1.l_true = B B_1.l_false = B.code = B_1.
$B \rightarrow E_1$ **rel** E_2	B.code = E_1.
$B \rightarrow$ **true**	B.code= *ger*
$B \rightarrow$ **false**	B.code= *ger*

Wenn B_1 == **true**, dann B = **true** (ohne dass B_2 ausgewertet wird)
B_1 und B haben in diesem Fall das gleiche Sprungziel

Wenn B_1 == **false** , muss auch B_2 ausgewertet werden.
D.h. man muss an den Anfang von B_2 springen.

Wenn nun B_2 == **true** (bzw. B_2 == **false)**, dann ist auch B == true
(bzw. B == false) weil man B2 nur auswertet, wenn B1 == false)

Also: beide Sprungziele werden einfach vererbt

B2.l_true = B.l_true

B2.l_false = B.l_false

Sprung-Code für boolsche Ausdrücke

Produktion	Semantische Regel
$B \rightarrow B_1 \,||\, B_2$	B_1.l_true || B.l_true B_1.l_false = **new** Label() B_2.l_true = B.l_true B_2.l_false = B.l_false B.code = B_1.code || B_1.l_false || B_2.code
$B \rightarrow B_1$ && B_2	B_1.l_true= **new** Label() B_1.l_false = B.l_false B_2.l_true = B.l_true B_2.l_false = B.l_false B.code = B_1.code || B_1.l_true || B_2.code
$B \rightarrow$! B_1	B_1.l_true = B.l_false B_1.l_false = B.l_true B.code = B_1.code
$B \rightarrow E_1$ **rel** E_2	B.code = E_1.code || E_2.l_true || *gen* ('if' E1.addr **rel**.op E2.addr 'goto' B.l_true) || *gen* ('goto' B.l_true)
$B \rightarrow$ **true**	B.code= *gen* ('goto' B.l_true)
$B \rightarrow$ **false**	B.code= *gen* ('goto' B.l_false)

Sprung-Code für boolsche Ausdrücke

Produktion	Semantische Regel
$B \rightarrow B_1 \,||\, B_2$	B_1.l_true || B.l_true B_1.l_false = **new** Label() B_2.l_true = B.l_true B_2.l_false = B.l_false B.code = B_1.code || B_1.l_false || B_2.code
$B \rightarrow B_1$ && B_2	B_1.l_true= **new** Label() B_1.l_false = B.l_false B_2.l_true = B.l_true B_2.l_false = B.l_false B.code = B_1.code || B_1.l_true || B_2.code
$B \rightarrow$! B_1	B_1.l_true = B.l_false B_1.l_false = B.l_true B.code = B_1.code
$B \rightarrow E_1$ **rel** E_2	B.code = E_1.cod || *gen* ('goto' B.l_true)
$B \rightarrow$ **true**	B.code= *gen* ('goto' B.l_true)
$B \rightarrow$ **false**	B.code= *gen* ('goto' B.l_false)

B hat gleichen Code wie B_1, nur Sprungziele sind vertauscht

Sprung-Code für boolsche Ausdrücke

Produktion	Semantische Regel
$B \rightarrow B_1 \,||\, B_2$	B_1.l_true || B.l_true B_1.l_false = **new** Label() B_2.l_true = B.l_true B_2.l_false = B.l_false B.code = B_1.code || B_1.l_false || B_2.code
$B \rightarrow B_1$ && B_2	B_1.l_true= **new** Label() B_1.l_false B_2.l_true = B_2.l_false B.code = B_1.code || B_1.l_true || B_2.code
$B \rightarrow$! B_1	B_1.l_true = B.l_false B_1.l_false = B.l_true B.code = B_1.code
$B \rightarrow E_1$ **rel** E_2	B.code = E_1.code || E_2.l_true || *gen* ('if' E1.addr **rel**.op E2.addr 'goto' B.l_true) || *gen* ('goto' B.l_true)
$B \rightarrow$ **true**	B.code= *gen* ('goto' B.l_true)
$B \rightarrow$ **false**	B.code= *gen* ('goto' B.l_false)

E_1.code und E_2.code liefern ihre Resultate in E_1.addr und E_2.addr ab.

Man muss die Inhalte beider Adressen vergleichen (mit rel.op)

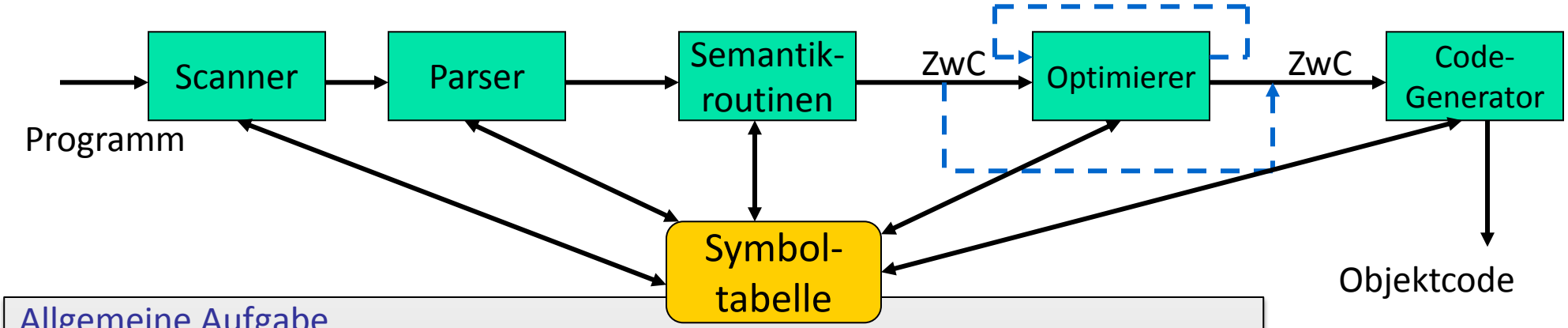
Position

- 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
- 6.10
  Typprüfung

Einsatz der Symboltabelle



Allgemeine Aufgabe

- Verbindung lexikalischer Namen (Symbole) mit ihren Attributen
- zentrales Hilfsmittel für Vereinbarung und Nutzung von Symbolen

Was soll in der Symboltabelle gespeichert werden?

- Variablennamen
- …
- Prozedur- und Funktionsnamen
- …
- Sprungadressen (Labels)
- …

Achtung: eine Symboltabelle ist eine Datenstruktur, die **nur** zu Übersetzungszeit existiert

Rolle der Symboltabelle

Welche Art von Informationen braucht ein Compiler?

- Namen (Zeichenkette)
- Typ (Speicherplatzerfordernis, Operatoranwendbarkeit)
- Dimensionsinformation (Felder)
- Prozedur-/ Funktionsdeklaration
- Anzahl und Typ der Argumente von Prozeduren/Funktionen
- Gültigkeitsbereich von Deklarationen
  (Blockstruktur→ verschachtelte Gültigkeitsbereiche)
- Sichtbarkeitsbereich von Symbolen (Modul-, Klassenkonzept)
- Überladungsinfos für Operatoren
- Speicherklasse (static, extern, global, ...)
- Offset im Speicher
- falls Name eines Records: dann Strukturbeschreibung
- falls Parameter: call-by-value, call-by-reference, …

Beschränkung (Vorlesung):

blockstrukturierte Symboltabellen verwendet von Compilern für
- Pascal,
- C, C++,
- Modula,
- Java, …

Inhalt der Symboltabelle

Symboltabelle assoziiert Namen/Symbole

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

Alternative Implementierung von Symboltabellen

Fall: Einpass-Compiler

- Variante 1
  Knoten, die Blockstrukturen eröffnen, erhalten individuelle Symboltabellen ihrer Blöcke (als Liste von Bezeichnern in Form von Hash-Tabellen)
- Variante 2
  Kellerspeicher mit Blockmarkierungen als eleganter Ansatz

- lineare Liste
  - Komplexität: **O(n)** Tests pro Suchoperation
  - einfach zu erweitern, keine feste Größe
  - ein neuer Eintrag pro neu eingefügtes Elemer
- geordnetes lineares Feld
  - Komplexität: **$O(\log_2 n)$** Tests pro Suchoperation mit binärer Suche
  - Einfügen ist aufwendig, da Reihenfolge erhalten bleiben muss
- 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 (abhängig vom Füllstand)
  - Einfügen ist unterschiedlich schwierig (abhängig vom gewählten Verfahren)

Kellerimplementation

- Aufbau
  - Tabelle besteht aus **Blöcken**, die auf dem Stack liegen
  - beim Betreten eines Blockes im Quell-Programm
    wird ein Block in der Symboltabelle angelegt, d.h. auf dem Keller angelegt
  - beim Verlassen eines Blockes im Programm wird der letzte Block gelöscht
  - Bezeichner werden nur im aktuellen Block eingetragen
    (auf bereits deklarierte Bezeichner wird nur im aktuellen Block geachtet)
  - beim Suchen eines Bezeichners, werden die Blöcke ausgehend vom letzten zum ersten Eintrag hin untersucht, der erste Treffer wird gewählt
  - Sonderfall: Prozedur-/Funktionsparameter, Rekords

- Feststellung

  Keller ist am Ende des Syntaxanalyse leer
  (Kellerimplementierung ist ungeeignet für Mehrpass-Compiler)

Blockstrukturierte Symboltabelle

Verwaltung von **Gültigkeitsbereichen**!!

- wodurch entstehen Gültigkeitsbereiche?
  Definition eines Moduls/Programms, einer Funktion/Prozedur/ eines Blocks

- Anfragen zur Analysezeit nach einem Namen,
  a) muss die in diesem Gültigkeitsbereich gültige Deklaration mit ihren Informationen zurückgegeben werden, **oder**
  b) Deklaration aus einem "äußeren" Gültigkeitsbereich
  (innerer Gültigkeitsbereich überschreibt möglicherweise Deklarationen in äußeren Bereichen)

Block

- Gültigkeitsbereich von Bezeichnern
- verschachtelt
- Zur Strukturierung von
  a) Namensraum eines Programms
  b) Symboltabelle

→

Aufbau der Symboltabelle

- Folge von Bezeichnerblöcken
- Bezeichnerblock::= alle Bezeichner eines Blocks mit Attributen
  - Sprache C: 2-stufig
  - Pascal: unbeschränkt (real 10)

Deklarationsniveaus: beliebig

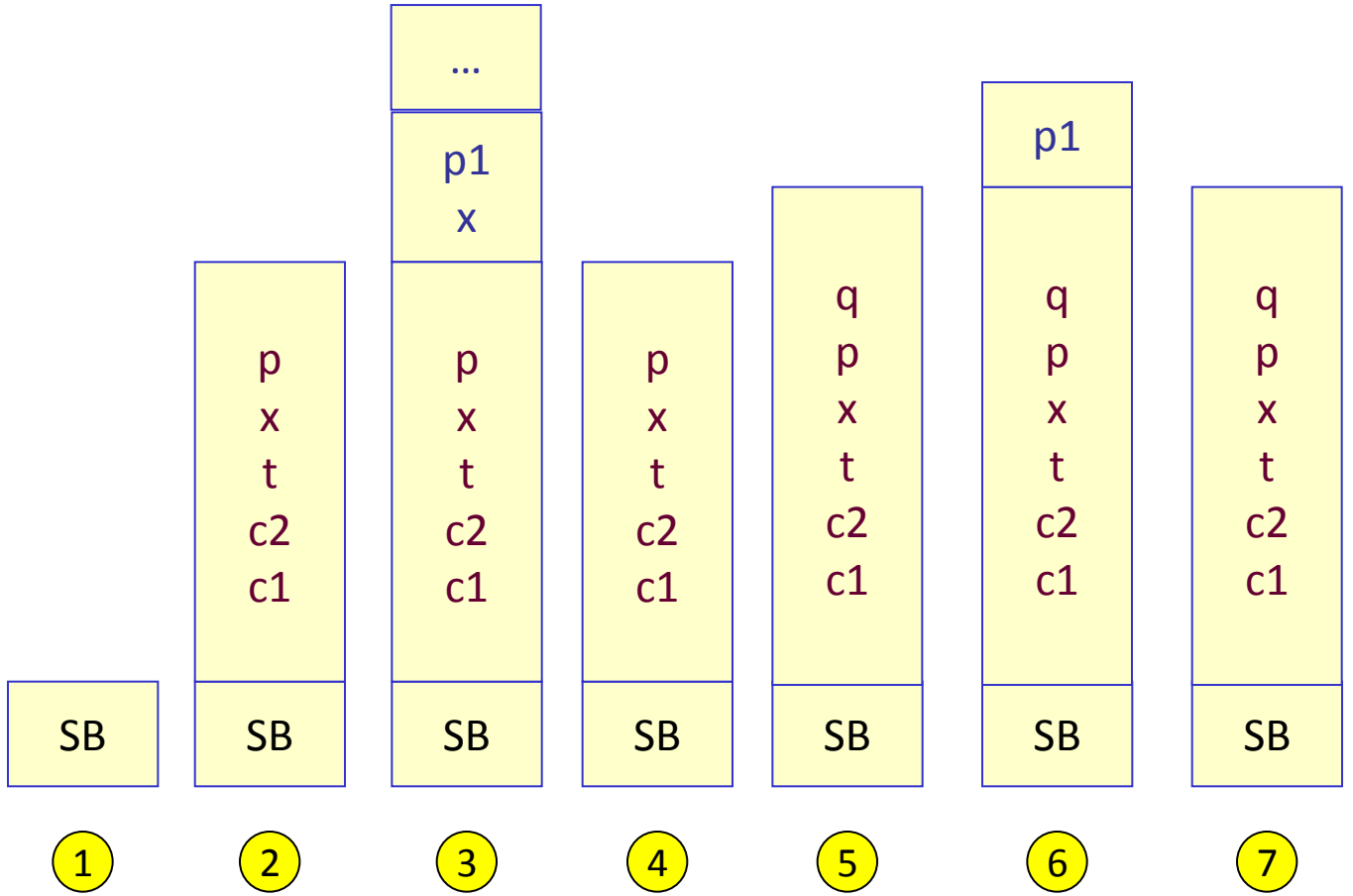
Prozeduren/ Funktionen

Blockstrukturierte Symboltabelle: Beispiel

```
1  program Prog;
       const  c1= 100;
              c2= 3.14;
       type   t= boolean;
       var    x: integer;

       procedure p ();
2             var x : integer;
              procedure p1
3             begin  ...
              end { p1 } ;
       begin
              x := c1;
4      end { p } ;
5      procedure q ();
6             const p1= 20;
       begin  ...
7      end { q } ;
begin
       ...
8  end { Prog }
```

dynamische Tabellenverwaltung



SB= Standardbezeichner

Operationales Interface: Symboltabelle

Realisierung als Hash-Tabelle

Eigenschaften von Gültigkeitsbereich (GB)

- neue Deklarationen können nur im aktuellen 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

Operationales Interface: Symboltabelle

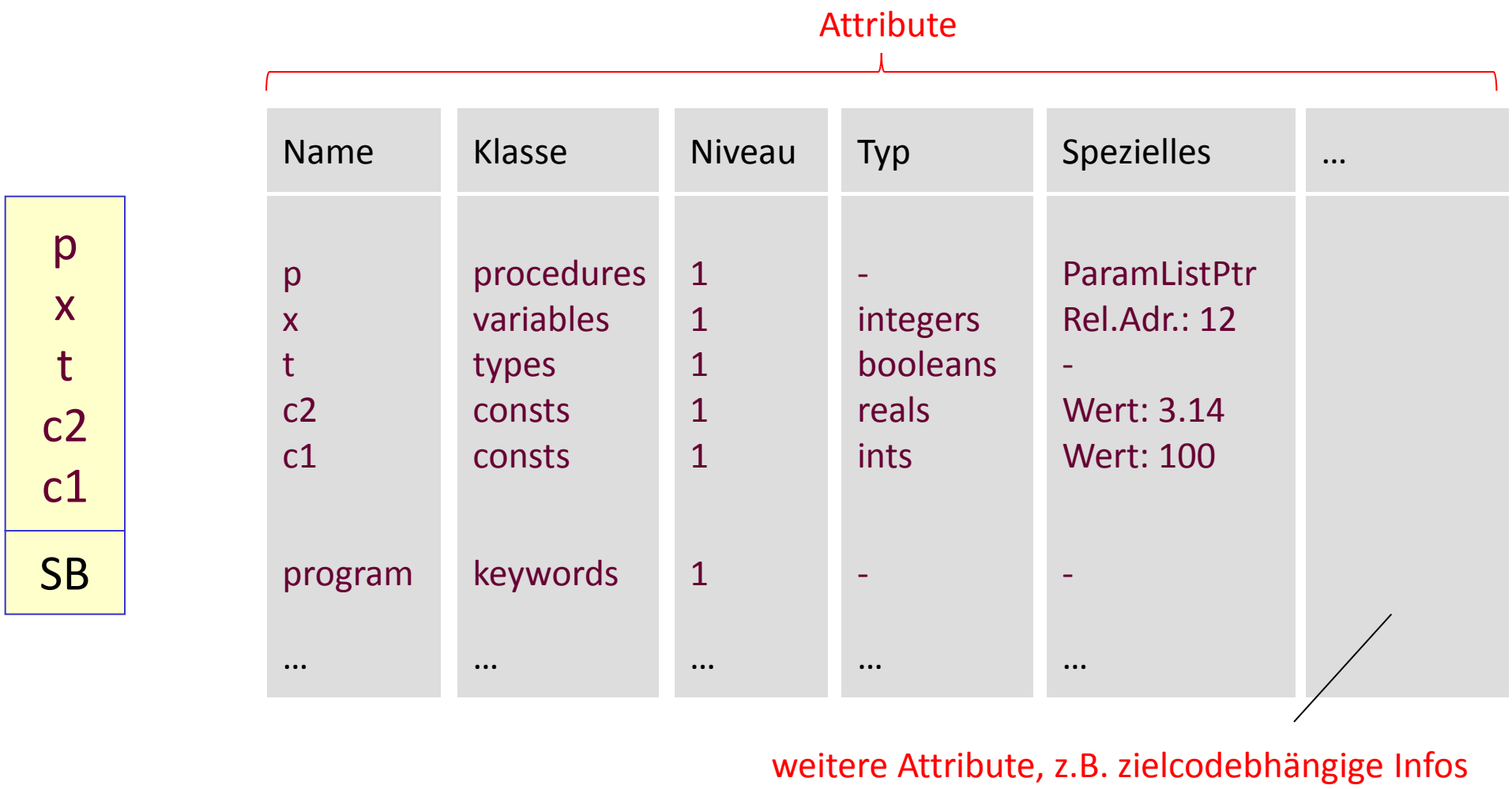
- void initSymbolTable()
- void beginScope()
- void endScope()
- idPtr putSymbol(string s, idClass cl)
- idPtr getSymbol(string s)
- int currentScope()

- … name(int entry)
- … class(int entry)
- … type(int entry)
- … scope(int entry)
- …
- Boolean isStandard(int entry)

} Attribut-Zugriff

Achtung Interface
der Symboltabelle mit kompletten Signature
wird in den Praktikumsübungen
vorgestellt !!!

Symboltabelle mit Codegenerierungsinfos



Erweiterung der blockstrukturierten Symboltabelle

- Zuordnung von Speichergrößen für Namen mit einem Typ
- relative Adressen eines Namens
  lassen sich dann als Offsets in Bezug zum Beginn des Datenbereiches ermitteln

Symboltabelle

Namen mit Typinfos und relativen Adressen

bei Speicherbedarfsermittlung von Größen, die der Compiler für die Codegenerierung benötigt

bislang für Codegenerierung noch ungelöst:
endgültiges Speicherlayout wird durch
Adressierungsconstraints der jeweiligen Zielmaschine eingeschränkt

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
  Überblic
- 6.2
  Attribut
- 6.3
  S-attribu
- 6.4
  Attributi
  und erer
- 6.5
  L-attribu
- 6.6
  Verfahre
  Überblic
- 6.7
  Entwurf
- 6.8
  Drei-Adress-Code-Generierung (einige Aspekte)
- 6.9
  Symboltabelle
- 6.10
  Typprüfung

- 6.10.1
  Typsysteme, Typchecker
- 6.10.2
  Typchecker für eine einfache Sprache
- 6.10.3
  Wandlung kompatibler Typen
- 6.10.4
  Überladung von Operatoren und Funktionen
- 6.10.5
  Typcodierung

Statische und dynamische Typprüfung

zur Ausführungszeit: dynamische Überprüfung

- prinzipiell kann jede Überprüfung dynamisch erfolgen,
  wenn Zielcode
  - den Typ des Elements zusammen mit
  - dem aktuellen Wert verwaltet

  (aber: schlechte Laufzeit- und Speichereigenschaften)

zur Übersetzungszeit: statische Überprüfung

- streng-getypte Sprachen erlauben (nahezu) komplette Typprüfung
  bereits durch den Compiler

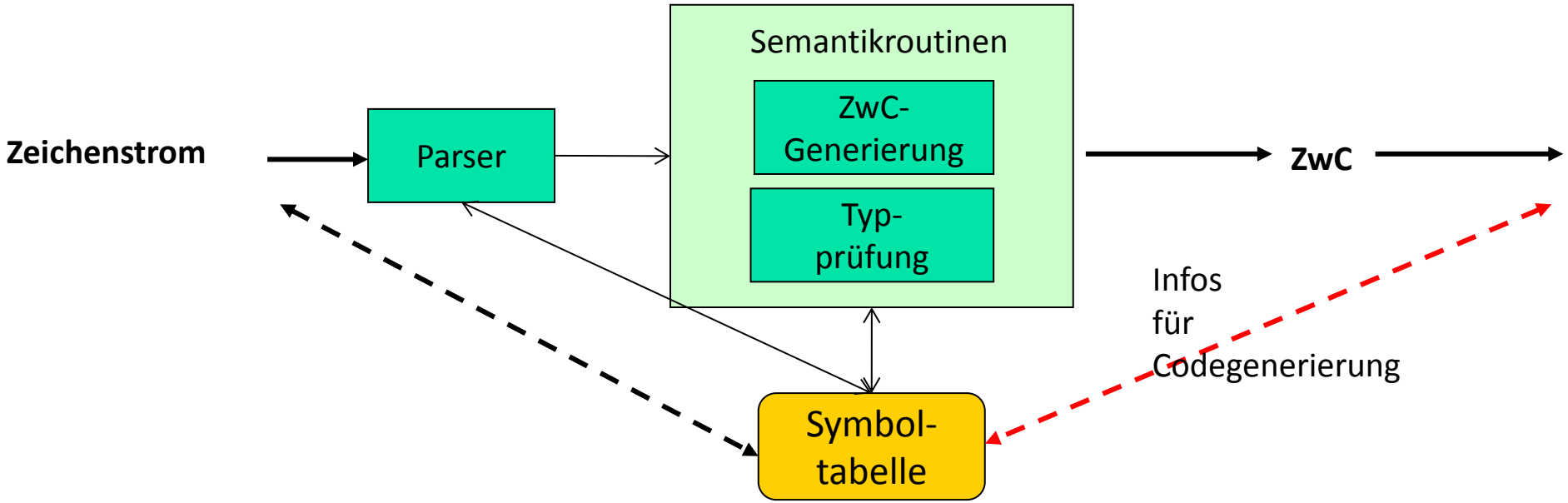
notwendige dynamische Überprüfungen

- aber auch bei streng-getypten Sprachen

  z.B. Indexverletzung

  (Grenzüberprüfung von Feldern bzw. Feldzugriffen)

Syntaxbaumaufbau und Typprüfung



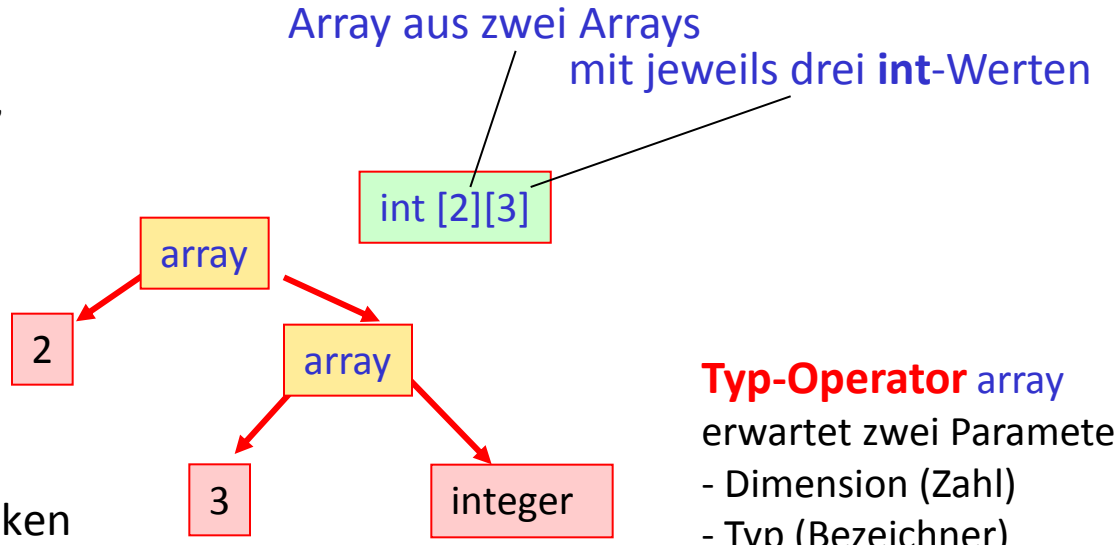
Typchecker

- **allgemein** ein Übersetzungsschema,
  das den Typ jedes Ausdrucks aus den Typen seiner Teilausdrücke bestimmt
- **speziell** ein Analyseprogramm zur Behandlung von Arrays, Zeigern, Anweisungen und Funktionen

- bei komplexen Sprachen
  separater Pass für Typprüfung
  (z.B. Ada, SDL)
- bei einfachen Sprachen
  Kombination von Typprüfung mit Aufbau des Zwischencodes
  (z.B. Pascal)

Typbeschreibungen

- Typen haben eine Struktur, die durch **Typausdrücke** formal beschrieben sind



Typ-Operator array
erwartet zwei Parameter:
- Dimension (Zahl)
- Typ (Bezeichner)

- zwei Arten von Typausdrücken

Basistypen
boolean, char, integer,
real, …, void

konstruierte Typen
gebildet per Typoperator (Typkonstruktor)

Typausdrücke können mit Namen versehen werden
(Typbezeichner sind Typausdrücke)

… durch Anwendung von Typ-Konstruktoren auf Typausdrücke
(z.B. array, pointer, record, class, …)

Typkonstruktoren (1)

- **Felder**

  Ann.: I und T sind Typausdrücke,
  so beschreibt array(I, T) ein Feld von T, indiziert über I

  meist gibt es Einschränkung für den Indextyp

- **Produkte**

  Ann.: T1 und T2 sind Typausdrücke,
  so beschreibt T1 × T2 ein kartesisches Produkt über die Typausdrücke T1 und T2
  (Anwendung für Parameterlisten)

  × ist linksassoziativ

- **Records**

  Felder (Elemente) haben - im Unterschied zu Produkten - Namen
  z.B.: record ((a × integer) × (b × real))

Beispiele für Typkonstruktoren (2)

- **Zeiger**

  ist T ein Typausdruck,
  dann beschreibt pointer (T)
  den Typ »Zeiger auf ein Objekt vom Typ T«

  z.B. Pascal: var p: ^row

- **Funktionen**

  D → R beschreibt den Typ einer Funktion,
  die Werte des Typs D auf Werte des Typs R abbildet,

  z.B. integer × char → pointer(integer)

  → ist der Funktionskonstruktor

Eine praktische Art der Darstellung von Typausdrücken
ist die Verwendung von Graphen (ZwC-Repräsentation kommt später)

Besonderheiten

- **Rekursive Typen**

  Beispiel:
  ```
  class Link {
      int info;
      Link next;
      ...
  }
  ```

- **Typschablonen (Typausdrücke mit Variablen)**

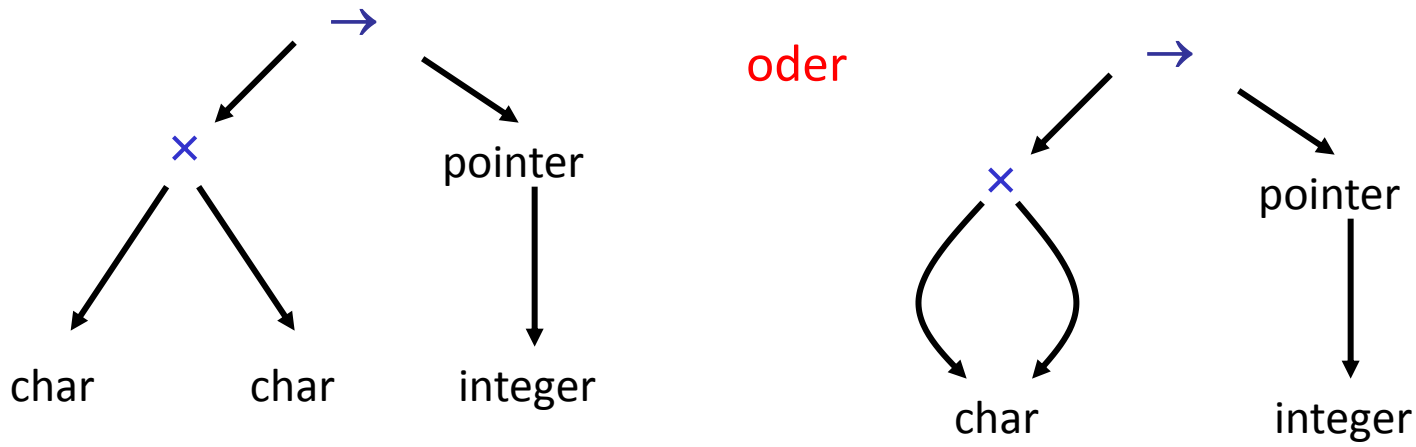
  z.B. Templates in C++, Typen mit Kontextparametern in SDL

Repräsentation von Typausdrücken

gerichteter azyklischer Graph (DAG),
der durch durch Syntaxanalyse entsteht:

- innere Knoten sind Typkonstruktoren,
- Blätter sind einfache Typen, Typbezeichner, Typvariablen

Beispiel: char × char → pointer(integer)



Deklaration und Speicherlayout

für Namen von Basis- und Array-Typen

D → T **id ;** D | ε

T → B C | **record { D }**

B → **int** | **float** Basistypen

C → ε | **[num]** C Array-Typen

Einführung synthetisierter Attribute
für jedes Nichtterminalsymbol:
type [Typinfo]
width [Speicher in Bytes]

und globale Größen t, w als Ersatz ererbter Attribute

Produktionen mit semantischen Regeln

T → B C	{t= B.type; w:= B.width}
B → **int**	{B.type:= integer; B.width:=4;}
B → **float**	{B.type:= float; B.width:=8;}
C → ε	{C.type:= t; C.width:=w;}
C → **[num]** C1	{C.type= array(num.value, C1.type); C.width:= num.value × C1.width; }

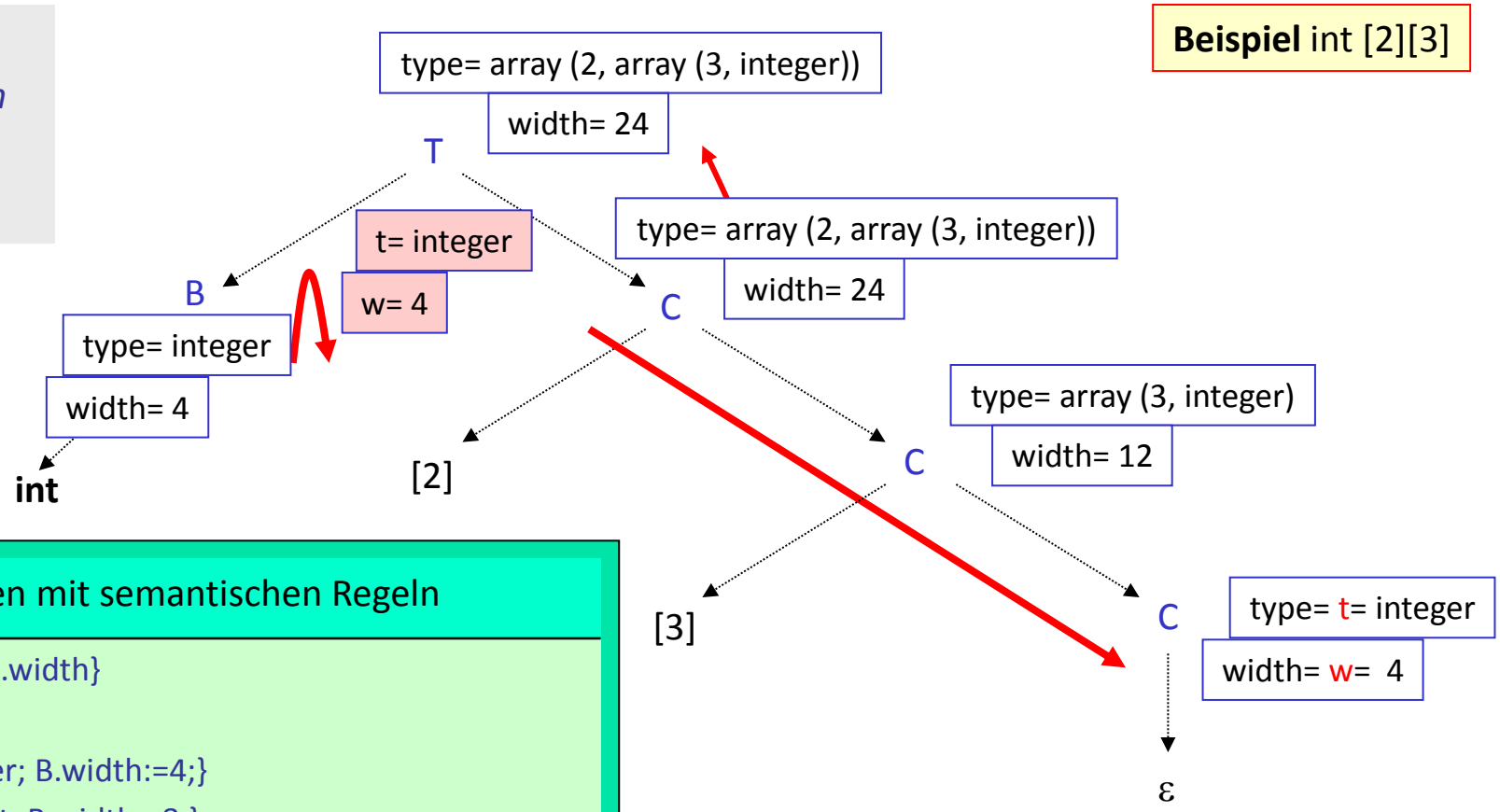
Achtung:

*semantische Operationen, die leider **nicht** alle am Ende auftauchen*

Ignorieren hierbei noch evtl. **Maschinenabhängigkeiten**:
Ausrichtung an Wortgrenzen entfällt (Beachtung in späterem Optimierungspass)

Deklaration und Speicherlayout

Einsatz ***globaler*** *Variablen anstatt ererbter Attribute*



Produktionen mit semantischen Regeln

T → B {t= B.type; w:= B.width}
C

B → **int** {B.type:= integer; B.width:=4;}

B → **float** {B.type:= float; B.width:=8;}

C → ε {C.type:= t; C.width:=w;}

C → **[num]** C1 {C.type:= array(num.value, C1.type);
C.width:= num.value × C1.width; }

Erweiterung des Übersetzungsschemas

Speicherlayout für Namen

betrachten jetzt Deklarationsliste

T als Typ, definiert wie vorher

Produktionen mit semantischen Regeln

$P \rightarrow$ {offset:= 0} D

$D \rightarrow$ T **id ;** {top.put (id.lexeme, T.type, offset); offset:= offset + T.width; } D

$D \rightarrow \varepsilon$

Offset-Bestimmung bei Symboltabelleneintrag

offset= relative Adresse des Speichers des bezeichneten Objektes im aktuellen Gültigkeitsbereich

(top= aktuelle Symboltabelle)

Produktionen mit semantischen Regeln

$P \rightarrow M\ D$

$M \rightarrow \varepsilon$ {offset:= 0}

$D \rightarrow$ T **id ;** {top.put (id.lexeme, T.type, offset); offset:= offset + T.width; } D

$D \rightarrow e$

***Achtung**:*

semantische Operationen, die leider auch ***nicht*** *am Ende auftauchen*

Erweiterung des Übersetzungsschemas:

Speicherlayout für Records (1)

Realisierung erfolgt über eigene (partielle) Symboltabelle je Record (!)

Produktionen mit semantischen Regeln

```
T → record { { Env.push(top);
               top:=new Env();
               Stack.push(offset);
               offset:=0; }

    D }      { T.type:= record(top);
               T.width:= offset;
               top= Env.pop();
               offset:= Stack.pop(); }
```

Achtung: Records können insbesondere verschachtelt sein

- Feldbezeichner eines Records müssen 1-deutig sein
- relative Adresse (offset) eines Feldbezeichners wird relativ zum Datenbereich des Records angegeben

→ strukturierte Namensräume

Erweiterung des Übersetzungsschemas:

Speicherlayout für Records (2)

Produktionen mit semantischen Regeln

```
T → record { { Env.push(top);
               top:=new Env();
               Stack.push(offset);
               offset:=0; }

   D }      {  T.type:= record(top);
               T.width:= offset;
               top:= Env.pop();
               offset:= Stack.pop(); }
```

Aktionen vor D

- **Retten** der existierenden Symboltabelle top auf einem SymboltabellenStack der Umgebung
- Anlegen einer neuen Symboltabelle (new Env)

- **Retten** des aktuellen Offsets auf einem (synchronen) Offset-Stack
- Intialisierung eines neuen Offsets für das aktuelle Record

- **Deklarationen** liefern Typen und Offsets, die in aktueller Symboltabelle vermerkt werden

Aktionen nach D

- **Erzeugung** eines Records unter Verwendung von der Symboltabelle top

- **Wiederherstellung** von Symboltabelle und Offsets

Typsysteme (Definition)

… sind Sammlungen von Regeln zur Zuweisung von Typausdrücken

Bem.:
in verschiedenen Compilern der **gleichen** Sprache können durchaus (in Nuancen) unterschiedliche Typsysteme zum Einsatz kommen

Spezifikation von Typsystemen

- syntaxgesteuerte Regeldefinition,

  so dass bekannte Techniken zur Implementation semantischer Aktionen eingesetzt werden können (attributierte Grammatiken)

- semantische Aktionen übernehmen Typcheck

  Sicherung einer partiellen Typäquivalenz

Einschränkung hinsichtlich Kompatibilität

Pascal: Array

Typ abhängig vom gewählten Indextyp

Typ unabhängig vom gewählten Indextyp

Aufgaben eines Typ-Checkers

Typ-Checker muss Typgleichheit / Typäquivalenz feststellen
Algorithmische Bestimmung der Typgleichheit hängt von Repräsentation der Typen ab

Typäquivalenz

1. Behandlung impliziter (Namensgebung) und expliziter Typnamen
2. Behandlung rekursiver Typen
3. Behandlung expliziter und impliziter Typwandlungen

zwei Typen sind strukturell äquivalent
gdw. eine der folg. Bedingungen erfüllt ist:

1. sie sind dieselben Basistypen
2. sie wurden durch Anwendung desselben Konstruktors auf strukturell äquivalente Typen gebildet

zwei Typen sind namens äquivalent, wenn sie namensgleich sind

Allgemeines Problem der Typäquivalenz

```
type link  = ↑cell; {link ist der Typname für ↑cell }
var  next   : link;
     last   : link;
     p      : ↑cell;
     q, r   : ↑cell;
```

Beispiel: Pascal-ähnliche Definition

Problem: sind next und last vom selben Typ wie p, q, r ?
haben p und q überhaupt denselben Typ ?

bei **Namensäquivalenz**

- next und last sind vom selben Typ
- p, q und r sind vom selben Typ
- p und next haben unterschiedliche Typen

bei **struktureller Typäquivalenz** müssten eigentlich alle Variablen jeweils denselben Typ haben
(aber Praxis: Ada, Pascal und Modula-2

- sehen unterschiedliche Typdefinitionen als unterschiedliche Typen an,
  somit hat hier trotz geforderter struktureller Typäquivalenz p einen anderen Typ als q und r

FRAGE: WARUM?

Berücksichtigung impliziter Typnamen

Für Pascal gilt:

enthält eine Deklaration einen Typausdruck, der kein Name ist, wird ein impliziter Name erzeugt:

```
type link  = ↑cell;

var  next   : link;
     last   : link;
     p      : ↑cell;
     q, r   : ↑cell;
```

```
type  link   = ↑cell;
      np1    = ↑cell;
      np2    = ↑cell;

var  next   : link;
     last   : link;
     p      : np1;
     q, r   : np2;
```

p und q sind nicht vom selben Typ

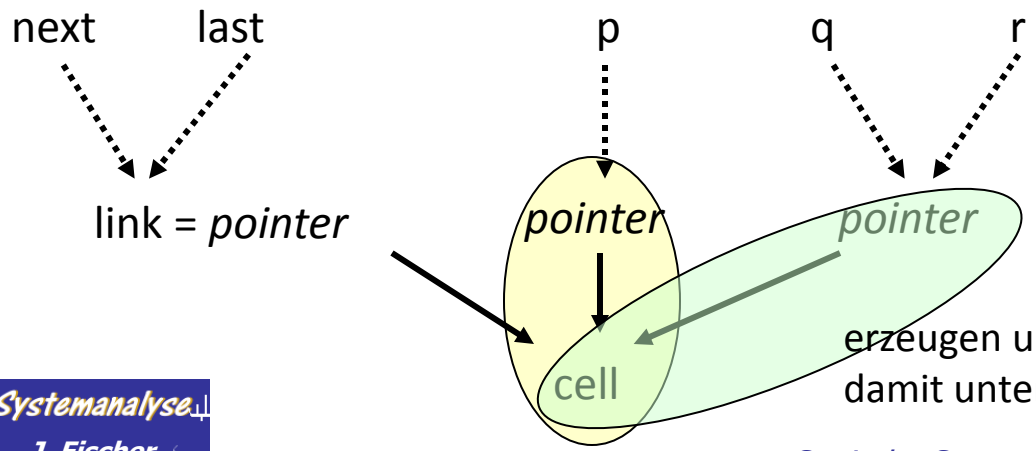
Typäquivalenz in Pascal

Typ-Ausdrücke sind **äquivalent**, wenn sie durch **denselben** Knoten im Graph repräsentiert werden

Typ-Graph: Struktur zur Übersetzungszeit

- jeder Konstruktor oder Basistyp erzeugt einen Knoten
- jeder Name erzeugt ein Blattknoten (der mit dem Typdeskriptor assoziiert wird)

in Pascal/Ada/ Modula: Deklaration von Bezeichnern mit impliziten Typnamen (immer dann, wenn ein Typausdruck verwendet wird)




```
type link  = ↑cell;
var  next  : link;
     last  : link;
     p     : ↑cell;
     q, r  : ↑cell;
```

erzeugen unterschiedliche implizite Typnamen und damit unterschiedliche Knoten

Fazit: Typausdrücke, Typprüfung und Typsystem

Basis

Compiler (**Typ-Checker)**

- weist jeder Komponente im Quellprogramm einen Typausdruck zu
- stellt fest, ob die Typausdrücke konform zu einer Menge von Regeln (Typsystem der Sprache) sind
- hat Potential für Fehlerbestimmung

Typ-Checker ist allgemein ein Übersetzungsschema,

das den Typ jedes Ausdrucks aus den Typen seiner Teilausdrücke bestimmt

und über semantische Regeln die Typtests realisiert

Typausdrücke

- **array** [256] **of char**
  führt zum Typausdruck: *array* ((1..256), *char*)
- ↑**int**
  führt zum Typausdruck: *pointer* (integer)

Typäquivalenz

Typ-Checker muss

(1) Namensäquivalenz

zwei Typen sind äquivalent, falls sie den gleichen Namen haben

jeder neue Typname stellt einen neuen (anderen) Typ dar

(2) Strukturelle Äquivalenz

zwei Typen sind äquivalent, falls sie die gleiche Struktur haben (nachdem sämtliche Typnamen durch Typausdrücke ersetzt worden sind)

- $s \equiv t$, falls s und t vom gleichen Basistyp sind
- $array(s_1,s_2) \equiv array(t_1,t_2)$, falls $s_1 \equiv t_1$ und $s_2 \equiv t_2$
- $s_1 \times s_2 \equiv t_1 \times t_2$, falls $s_1 \equiv t_1$ und $s_2 \equiv t_2$
- $pointer(s) \equiv pointer(t)$, falls $s \equiv t$
- $s_1 \rightarrow s_2 \equiv t_1 \rightarrow t_2$, falls $s_1 \equiv t_1$ und $s_2 \equiv t_2$

feststellen

Fazit: Typausdrücke, Typprüfung und Typsystem

Basis

Compiler (**Typ-Checker)**

- weist jeder Komponente im Quellprogramm einen Typausdruck zu
- stellt fest, ob die Typausdrücke konform zu einer Menge von Regeln (Typsystem der Sprache) sind
- hat Potential für Fehlerbestimmung

Typ-Checker ist allgemein ein Übersetzungsschema,

das den Typ jedes Ausdrucks aus den Typen seiner Teilausdrücke bestimmt

und über semantische Regeln die Typtests realisiert

Typausdrücke

- **array** [256] **of char**
  führt zum Typausdruck: *array* ((1..256), *char*)
- ↑**int**
  führt zum Typausdruck: *pointer* (integer)