

Übersicht

- Statische und dynamische Eigenschaften
- Konkrete und abstrakte Syntax
- Gültigkeit und Sichtbarkeit
- Aufgaben der semantischen Analyse
 - Identifizierung von Bezeichnern
 - Deklarationsanalyse
- Implementierung der Symboltabelle
- Typkonsistenz und Überladung von Bezeichnern
 - Typkonsistenz
 - Überladung
 - Polymorphismus

Lernziele

- Die wichtigsten Grundlagen und Aufgaben der semantischen Analyse benennen und erklären können, wie man diese prinzipiell löst
- Symboltabellen effizient implementieren können
- Wiedergeben können, wie man die Überladung von Bezeichnern auflöst

Aufgabe der semantischen Analyse

- Überprüfung bestimmter Programmeigenschaften, die nicht durch eine kontextfreie Grammatik beschreibbar sind (z.B. Kontextbedingungen)
- Unterschiedliche Arten von Programmeigenschaften
 - *Statisch* 🐡
 - *Dynamisch* 🐡

Beispiele (für Programmeigenschaften)

- Deklariertheitseigenschaften
 - Explizite Deklaration für jeden angewandt auftretenden Bezeichner
 - Keine Doppeldeklarationen
- Typkonsistenz
 - Übereinstimmung von Argumenttypen (von Operationen) mit Operandentypen

Grundlagen (aus der Sprach-Semantik)

- Gültigkeitsregeln
 - Legen für deklarierte Bezeichner fest, in welchem Programmteil die Deklaration einen Effekt hat
- Sichtbarkeitsregeln
 - Bestimmen, wo ein Bezeichner in seinem Gültigkeitsbereich sichtbar bzw. verdeckt ist

Beispiele (nicht-kontextfreie Sprachen)

- $\{w_c w_w \mid w \in (a|b)^*\}$
(vgl. def./angew. Vorkommen von Namen)
- $\{a^n b^m c^n d^m \mid n \geq 1 \wedge m \geq 1\}$
(vgl. Übereinstimmung Parameterzahl in Deklaration und Aufruf)

Statische semantische Eigenschaft (eines Programmkonstrukts)

- Für jedes Vorkommen des Konstrukts gilt
 - Der „Wert“ der Eigenschaft des Konstrukts ist in allen dynamischen Ausführungen derselbe
 - In korrektem Programm kann diese Eigenschaft zur Übersetzungszeit berechnet werden

Beispiel (statische Eigenschaft: Typ-Eigenschaft für arithmetische Ausdrücke)

- Annahme: Typ von terminalen Operanden, Variablen, Konstanten bekannt
- Dann: Beide Eigenschaften (für statische semantische Eigenschaften) erfüllt, d.h.
 - „Wert“ ändert sich nicht
 - Eigenschaft berechenbar

Operator	Typ des 1. Operanden	Typ des 2. Operanden	Resultattyp
+, -, *	int	int	int
	real	int	real
	int	real	real
	real	real	real
/	int real	int real	real
div	int	int	int

Dynamische semantische Eigenschaft (eines Programmkonstrukts)

- Eigenschaft ist nicht statisch
(d.h. „Wert“ ändert sich oder Eigenschaft nicht zur Übersetzungszeit berechenbar)
- Triviales Beispiel: Wert von Variablen

Beispiel (für nicht mehr statische Eigenschaft)

- Typ-Eigenschaft für arithmetische Ausdrücke nach Erweiterung um Potenz-Operator

Operator	1. Operand Typ, Größe	2. Operand Typ, Größe	Resultattyp
↑	int real	int > 0	int real (wie 1. Op.)
	int real ≠ 0	int = 0	int real (wie 1. Op.)
	int real ≠ 0	int < 0	real
	int real > 0	real	real
	int real = 0	real > 0.0	real (= 0.0)

Problem

- Typ von $e_1 \uparrow e_2$ hängt nicht nur von den Typen von e_1 und e_2 ab, sondern auch von der Größe ihrer Werte (ist also i.a. nicht statisch bestimmbar)

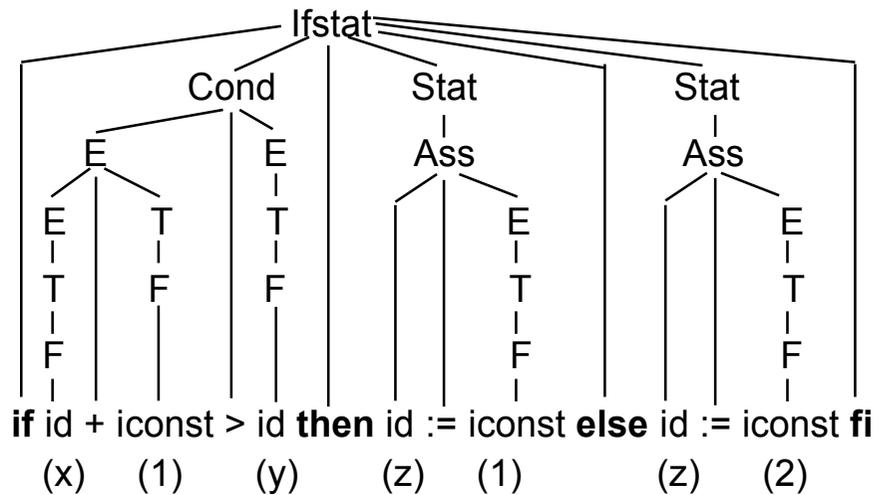
Konkrete und abstrakte Syntax

- Darstellung in **konkreter Syntax**
 - Baum gemäß der zugrunde liegenden kfG
- Darstellung in **abstrakter Syntax**
 - Baum der nur noch die für die Weiterverarbeitung wesentlichen Teile der Struktur enthält (insbesondere entfallen z.B. alle Terminalsymbole)

Für semantische Analyse meist benutzt

- Abstrakte Syntax

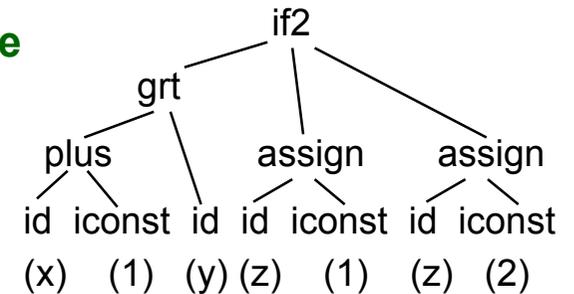
Konkrete Syntax



Programmstück

if x+1 > y then z := 1 else z := 2 fi

Abstrakte Syntax



Gültigkeits- und Sichtbarkeitsregeln

- Legen Beziehungen zwischen definierenden und angewandten Vorkommen von Bezeichnern fest
- Gültigkeitsbereich (eines definierenden Vorkommens eines Bezeichners x)
 - Teil des Programms, in dem sich ein angewandtes Vorkommen von x auf dieses definierende Vorkommen beziehen kann
- Sichtbarkeitsregeln
 - Schränken Gültigkeitsbereiche ein (z.B. Verdeckung globaler Bezeichner durch lokale Deklarationen)
 - Legen fest, auf welche definierende Vorkommen sich angewandte Vorkommen beziehen
- Hängen davon ab, welche Schachtelung von Scope-Konstrukten die Sprache erlaubt
 - Cobol: keine Schachtelung
 - Fortran: Schachtelungstiefe 1 (Prozedur-/Funktionsdeklarationen)
 - Algol60, Algol68, PL/1, Ada, Pascal: unbeschränkt tiefe Schachtelung (aber Unterschiede, s.u.)

Wichtigste Aufgabe (der semantischen Analyse)

*Prinzipielle Problematik
aus Kap. 2 bekannt*

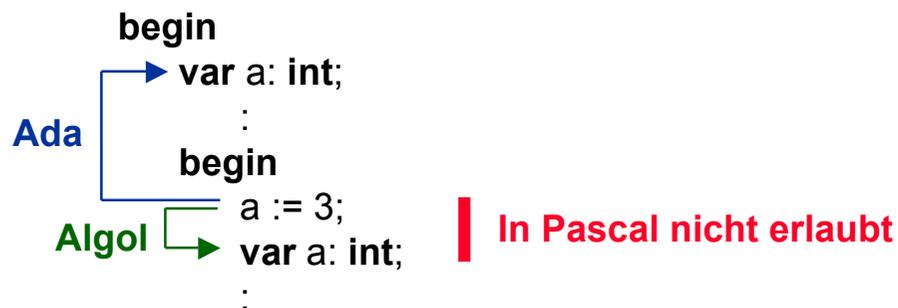
- Identifizierung von Bezeichnern, d.h. Zuordnung von angewandten Vorkommen zu definierenden gemäß Gültigkeits- und Sichtbarkeitsregeln

Unterschiedliche Gültigkeitsregeln

Keine Einpass-Übersetzung möglich

- Algol: Jeder in einem Block deklarierte Bezeichner ist im ganzen Block gültig
- Ada: Gültigkeitsbereich = Ende der Deklaration bis Ende des zugehörigen Blocks
- Pascal: Gültigkeitsbereich ist ganzer Block, aber kein angewandtes Vorkommen vor Ende der Deklaration erlaubt
- Zusätzlich: Möglichkeiten nicht direkt sichtbare Bezeichner sichtbar zu machen
 - Import bei getrennt übersetzten Programmteilen
 - Pascal: um Verbundnamen erweiterte Verbundkomponentennamen, **with**-statement
 - Ada: **use**-Direktive (macht Bezeichner umfassender Programmeinheiten sichtbar)
 - C++: Scope-Operator

Beispiel



Überprüfung der Kontextbedingungen

- Geg.: Programmiersprache mit geschachtelten Scope-Konstrukten
 - ohne Module
 - ohne Überladung ————— Wird später behandelt
- Wichtigste Teilaufgaben der semantischen Analyse (i.w. über *Symboltabelle* , Details s.u.)
 - *Deklarationsanalyse* 
 - Identifizierung von Bezeichnern
 - Prüfung der Deklariertheitseigenschaften
 - *Typkonsistenz* 
- Beispiele für weitere Aufgaben
 - Überprüfung des Kontrollflusses (z.B. existierende Fortsetzung bei **break** in C)
 - Überprüfung auf Eindeutigkeit (z.B. eindeutige Marken bei **case**-Anweisung)
 - Auf Namen bezogene Überprüfung (z.B. gleicher Name an Anfang und Ende eines Konstrukts)

Ziel der Identifizierung

- Herstellung des Bezugs von angewandten auf definierende Vorkommen
(wird von Typüberprüfung und Codeerzeugung benutzt)

Mögliche Darstellungsformen

- Bei jedem Knoten für ein angewandtes Vorkommen eines Bezeichners steht entweder
 - (1) Deklarative Information direkt
 - (2) Verweis auf den Knoten für die Deklaration
 - (3) Adresse in der Symboltabelle

Diskussion der Alternativen

- (1) und (2):
 - Symboltabelle nach Identifizierungsphase überflüssig
(Syntaxbaum bleibt einzige Datenstruktur)
- (2), zusätzlich: deklarative Information nur einmal festgehalten
 - Deshalb (im folgenden) Alternative (2)

Deklarationsanalyse (für Ada-ähnliche Gültigkeitsregeln)

I.w. depth-first-Baumdurchlauf

```
proc analyze_decl (k: node);  
  proc analyze_subtrees (root: node);  
    begin  
      for i := 1 to #descs(root) do (* #descs: Zahl der Kinder *)  
        analyze_decl (root.i) od (* i-tes Kind von root *)  
      end;  
    begin  
      case symb(k) of  
        block: begin  
          enter_block;  
          analyze_subtrees(k);  
          exit_block  
        end;  
        decl: begin  
          analyze_subtrees(k);  
          foreach dekl. Bezeich. id do enter_id(id, ↑k) od  
        end;  
        appl_id: (* angewandte Vorkommen eines Bezeichners id *)  
          speichere search_id(id) an k;  
        otherwise: if k kein Blatt then analyze_subtrees(k) fi  
      endcase  
    end  
  end  
end
```

*Hilfsfunktionen hier noch intuitiv;
weiter unten formal definiert*

Vermerkt das Öffnen eines neuen Blocks

*Setzt die Symboltabelle auf den Stand
vor dem letzten enter_block zurück*

*Fügt Eintrag für id in Symboltabelle ein;
2.Parameter enthält Verweis auf die
Deklarationsstelle von id*

*Sucht definierendes Vorkommen zu
id und gibt ggf. Verweis auf die
Deklarationsstelle von id zurück*

Ziel

- search_id-Funktion muss zu jedem Zeitpunkt die gemäß Sichtbarkeitsregeln richtige Deklaration auf **effiziente** Weise finden

Möglichkeiten (Aufwand jeweils in Abhängigkeit der Zahl der deklarierten Bezeichner)

- Lineare Liste, kellerartig organisiert (Aufwand linear)
- Binäre Suchbäume für jeden Block (Aufwand logarithmisch)
- Mischung aus (Hash-)Tabelle, linear verketteten Listen und Keller (Aufwand konstant)

Implementierung

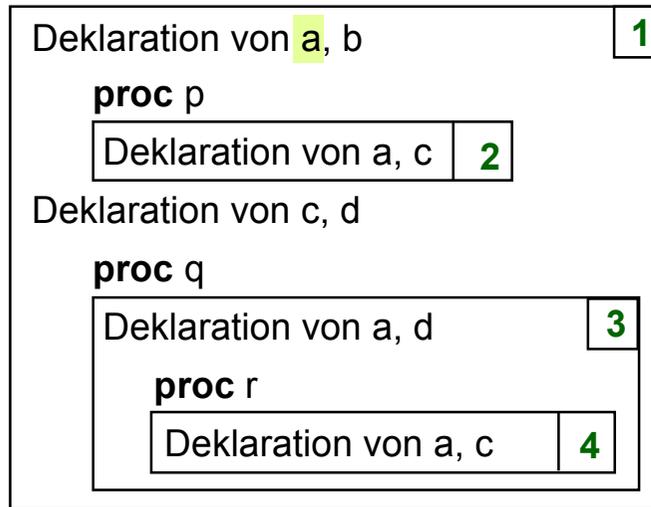
- Zu jedem (aktuell sichtbaren) definierenden Vorkommen (eines Bezeichners) linear verkettete Liste (der noch gültigen Vorkommen dieses Bezeichners) aus Elementen der Form

Block-Nr.	Verweis auf Unterbaum für die Deklaration	Verkettung der Bezeichner innerhalb eines Blocks	Verweis auf statischen Vorgänger
-----------	---	--	----------------------------------

- Zugriff auf die Bezeichner über Feld (indiziert mit Bezeichnern)
- Außerdem
 - Zum gleichen Block gehörende Bezeichner (rückwärts) verkettet
 - Kellerartig verwaltete Listenköpfe für die aktuell sichtbaren Blöcke, die auf die jeweils letzte Deklaration (und damit auf alle rückwärts verketteten Bezeichner) des Blocks zeigen

*Über Hashfunktion;
vom Scanner geliefert*

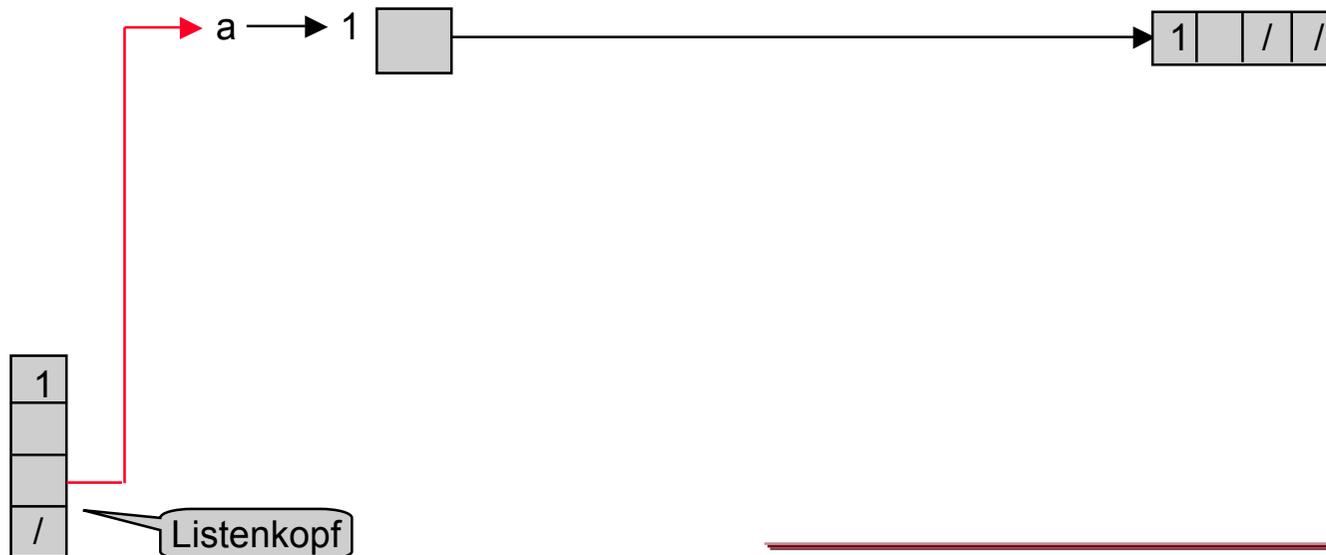
Beispiel



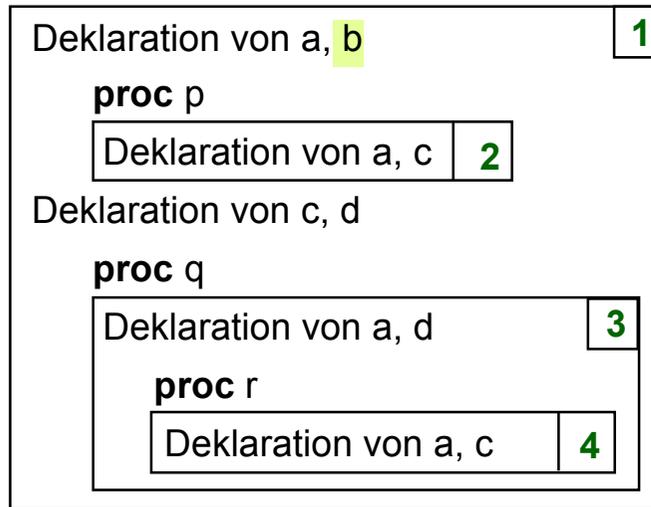
Abarbeitung der Deklarationen

```

create_symb_table
enter_block 1
enter_id(a, ↑(a in 1))
  
```



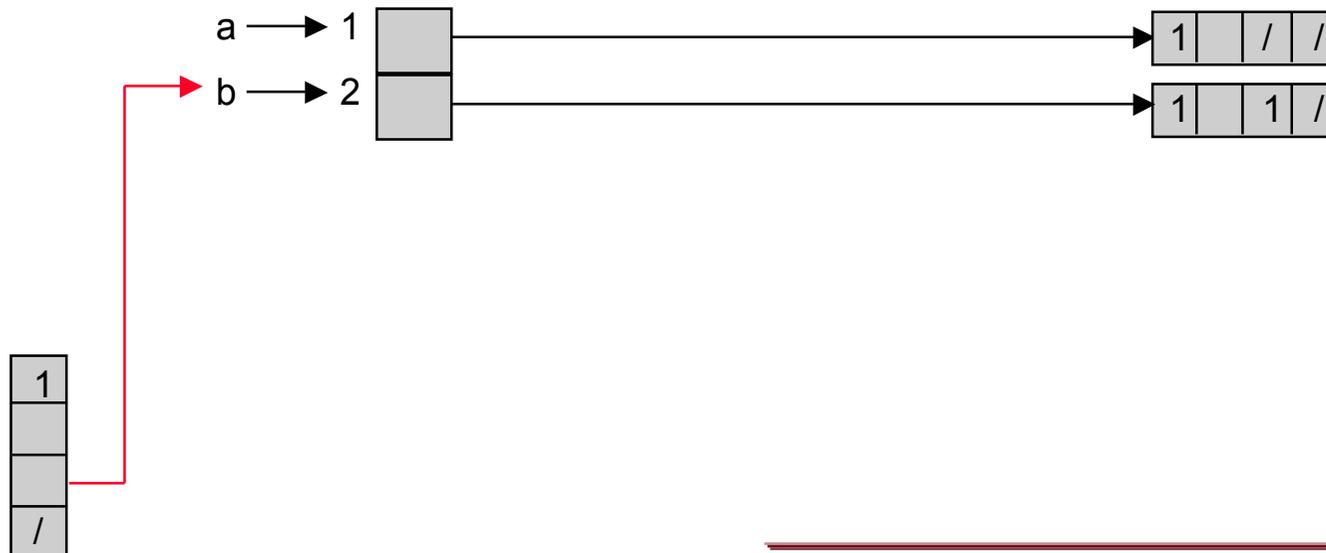
Beispiel



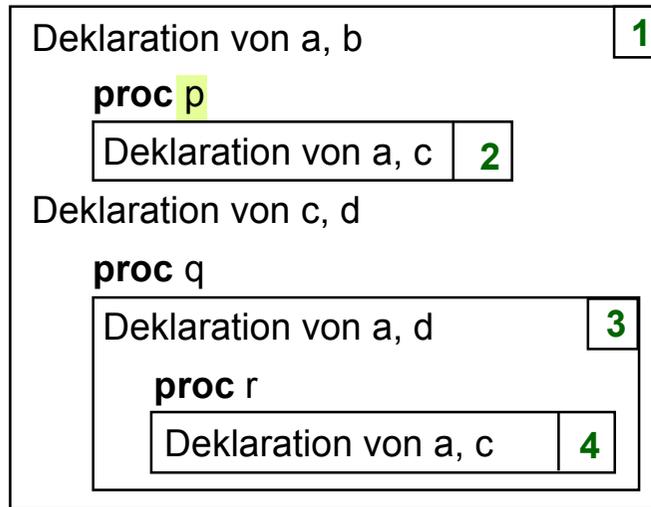
Abarbeitung der Deklarationen

```

create_symb_table
enter_block 1
enter_id(a, ↑(a in 1))
enter_id(b, ↑(b in 1))
  
```



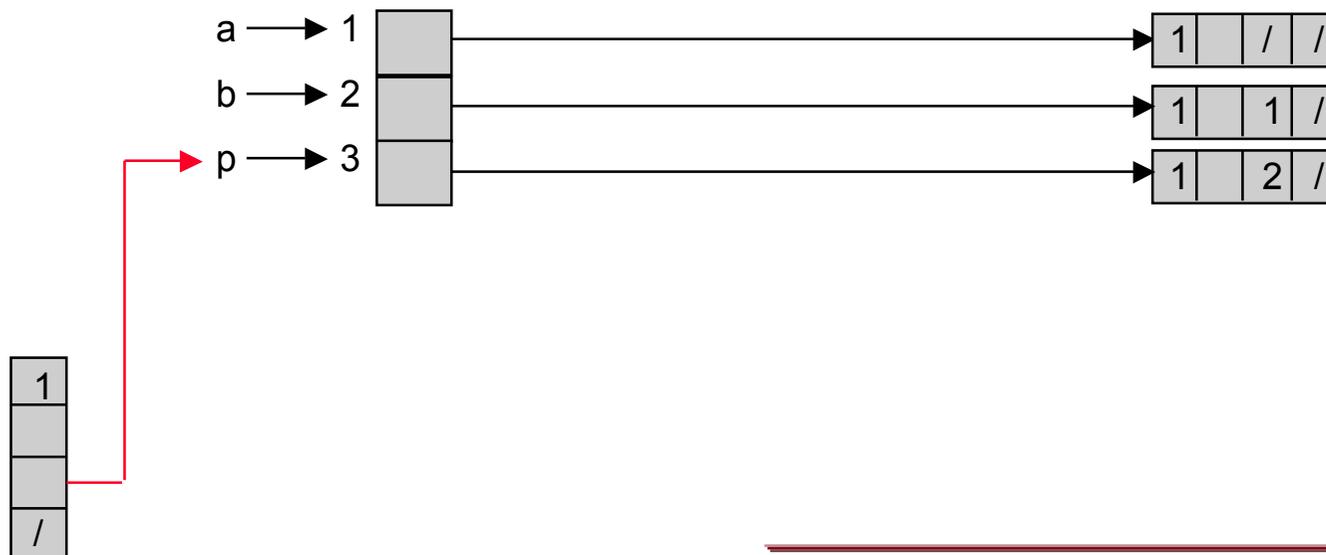
Beispiel



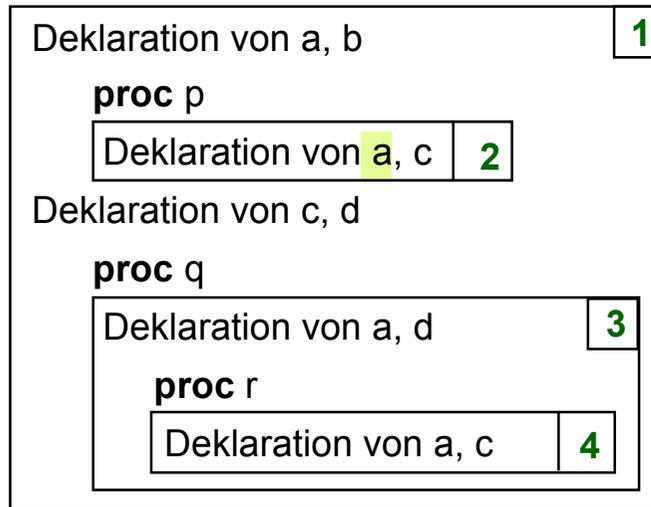
Abarbeitung der Deklarationen

```

create_symb_table
enter_block 1
enter_id(a, ↑(a in 1))
enter_id(b, ↑(b in 1))
enter_id(p, ↑(p in 1))
  
```



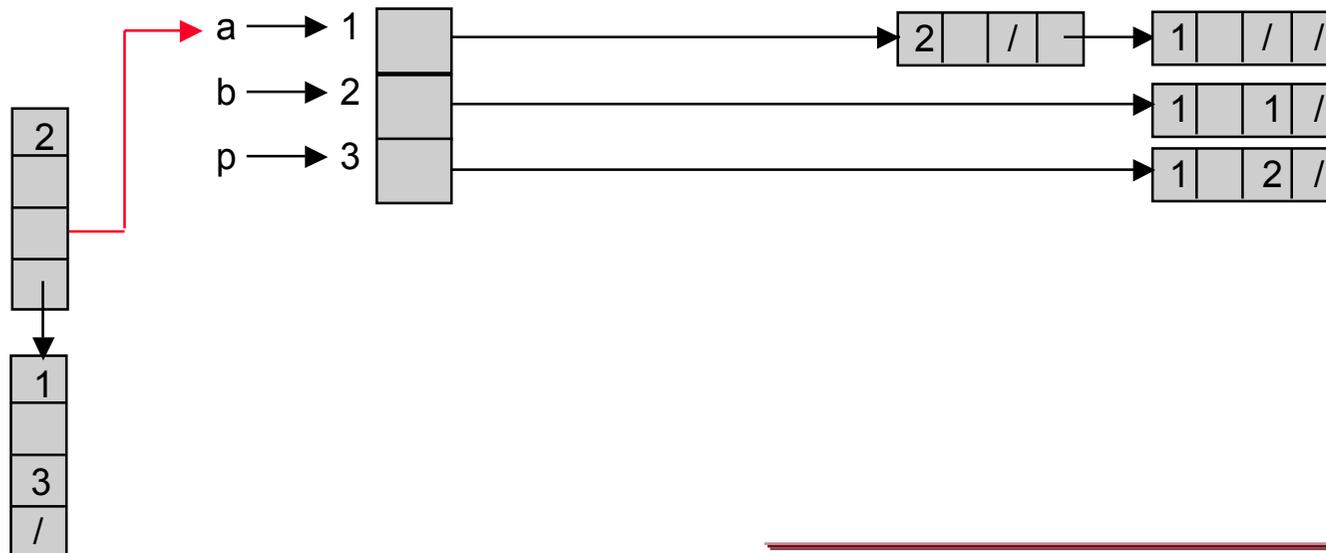
Beispiel



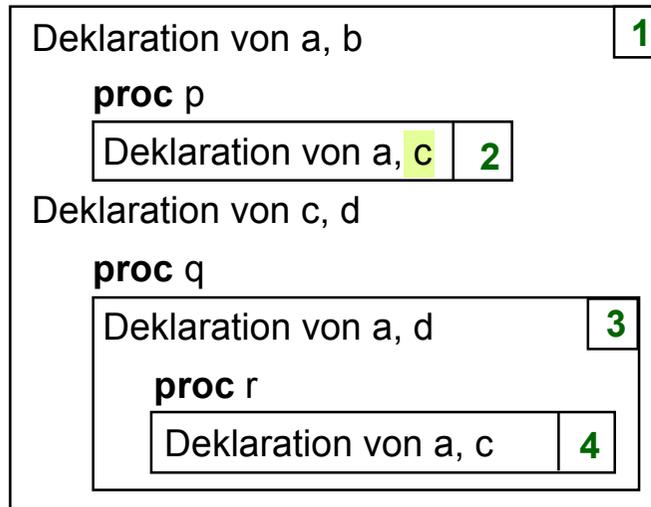
Abarbeitung der Deklarationen

```

create_symb_table
enter_block 1
enter_id(a, ↑(a in 1))
enter_id(b, ↑(b in 1))
enter_id(p, ↑(p in 1))
enter_block 2
enter_id(a, ↑(a in 2))
  
```



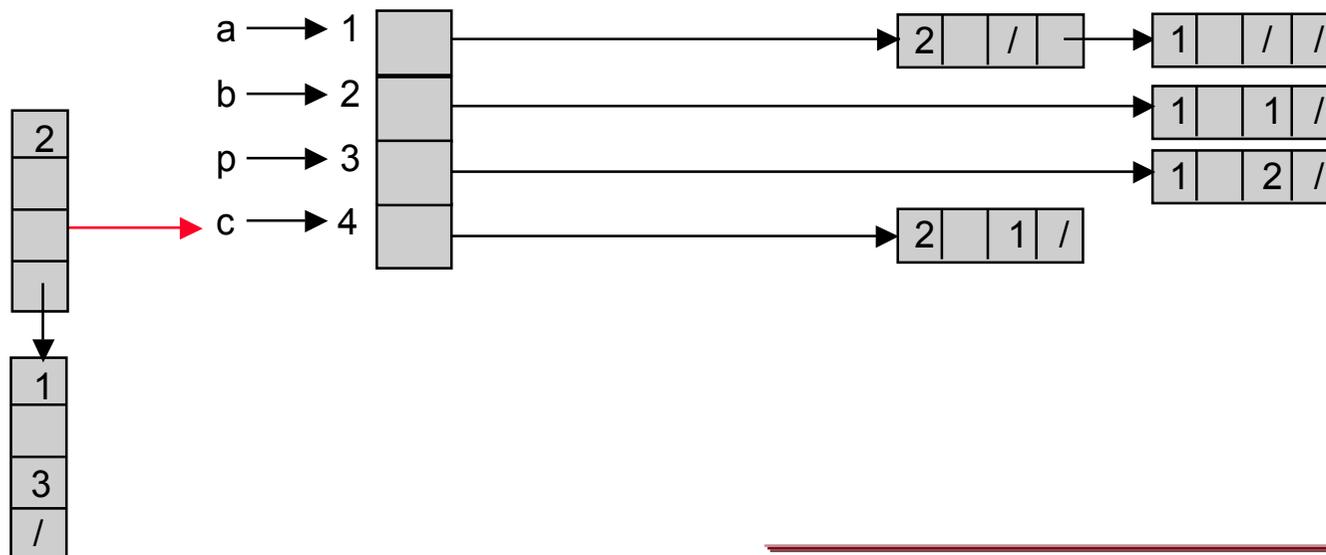
Beispiel



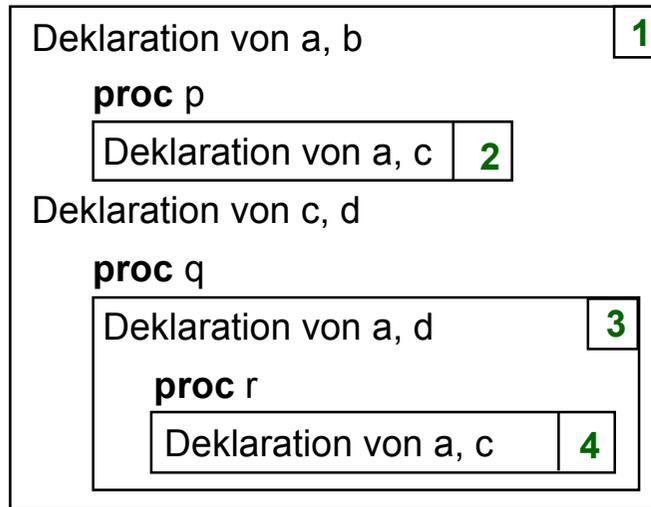
Abarbeitung der Deklarationen

```

create_symb_table
enter_block 1
enter_id(a, ↑(a in 1))
enter_id(b, ↑(b in 1))
enter_id(p, ↑(p in 1))
enter_block 2
enter_id(a, ↑(a in 2))
enter_id(c, ↑(c in 2))
  
```



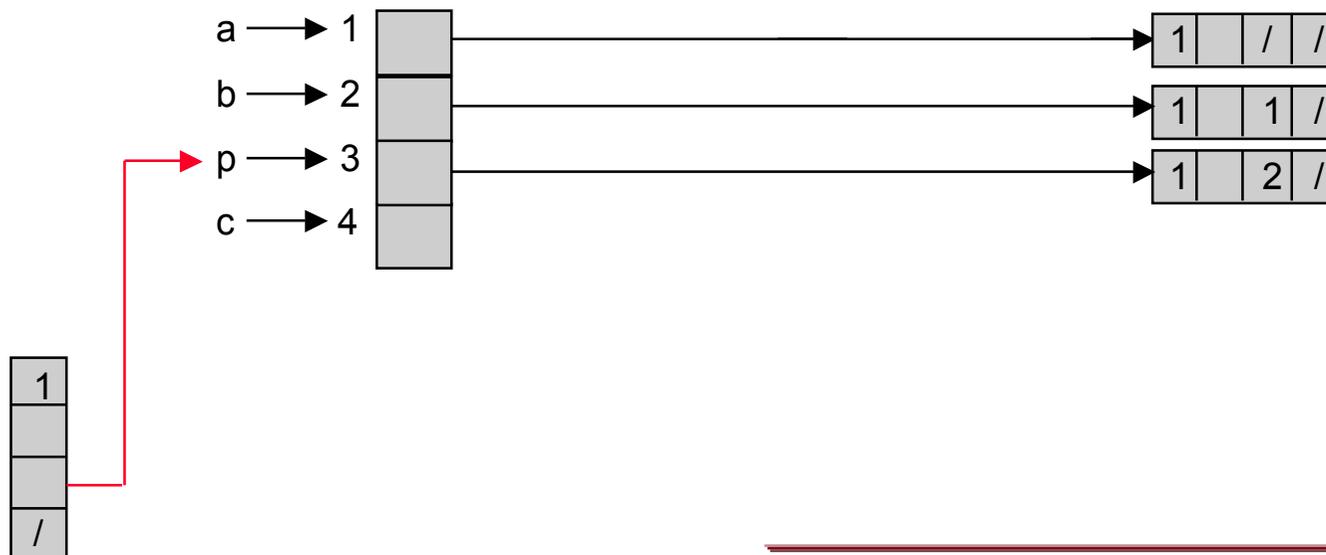
Beispiel



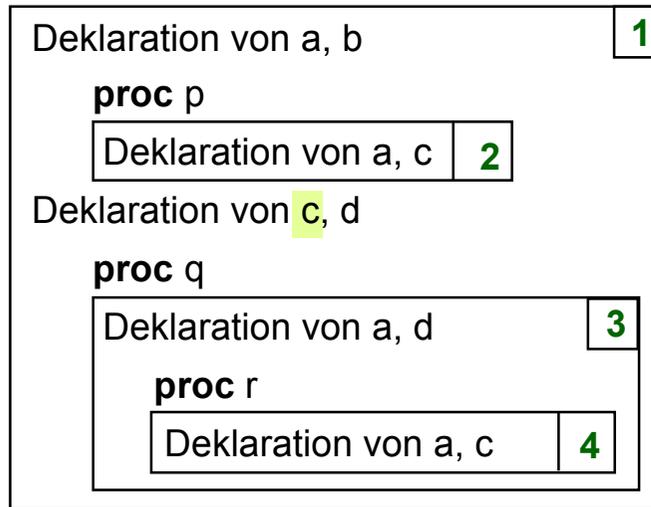
Abarbeitung der Deklarationen

```

create_symb_table
enter_block 1
enter_id(a, ↑(a in 1))
enter_id(b, ↑(b in 1))
enter_id(p, ↑(p in 1))
enter_block 2
enter_id(a, ↑(a in 2))
enter_id(c, ↑(c in 2))
exit_block 2
    
```



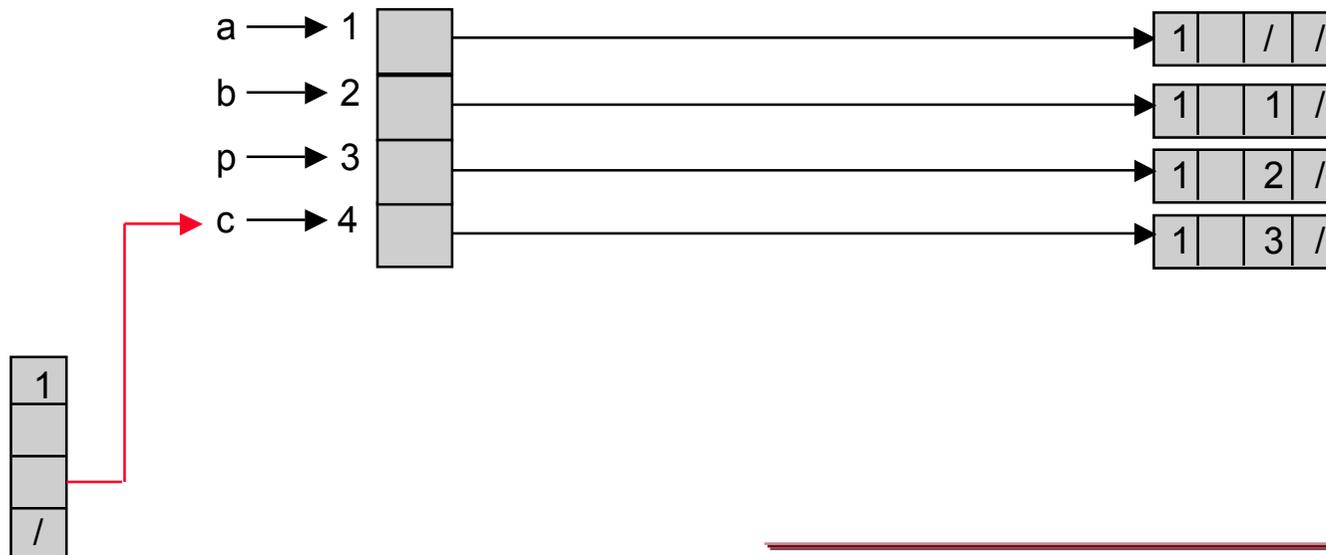
Beispiel



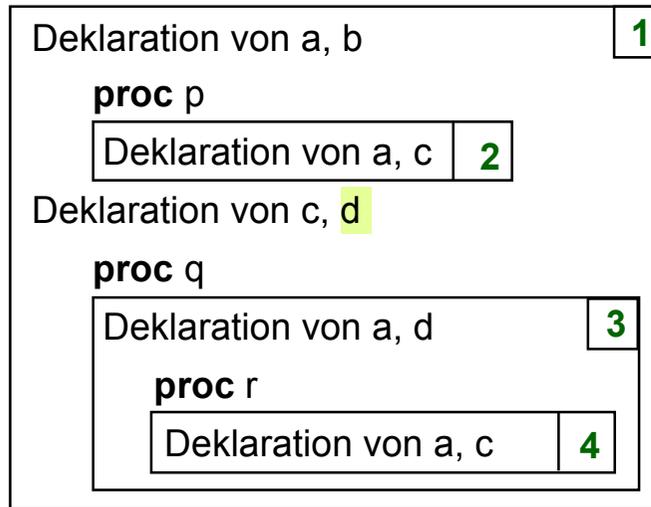
Abarbeitung der Deklarationen

```

create_symb_table
enter_block 1
enter_id(a, ↑(a in 1))
enter_id(b, ↑(b in 1))
enter_id(p, ↑(p in 1))
enter_block 2
enter_id(a, ↑(a in 2))
enter_id(c, ↑(c in 2))
exit_block 2
enter_id(c, ↑(c in 1))
    
```



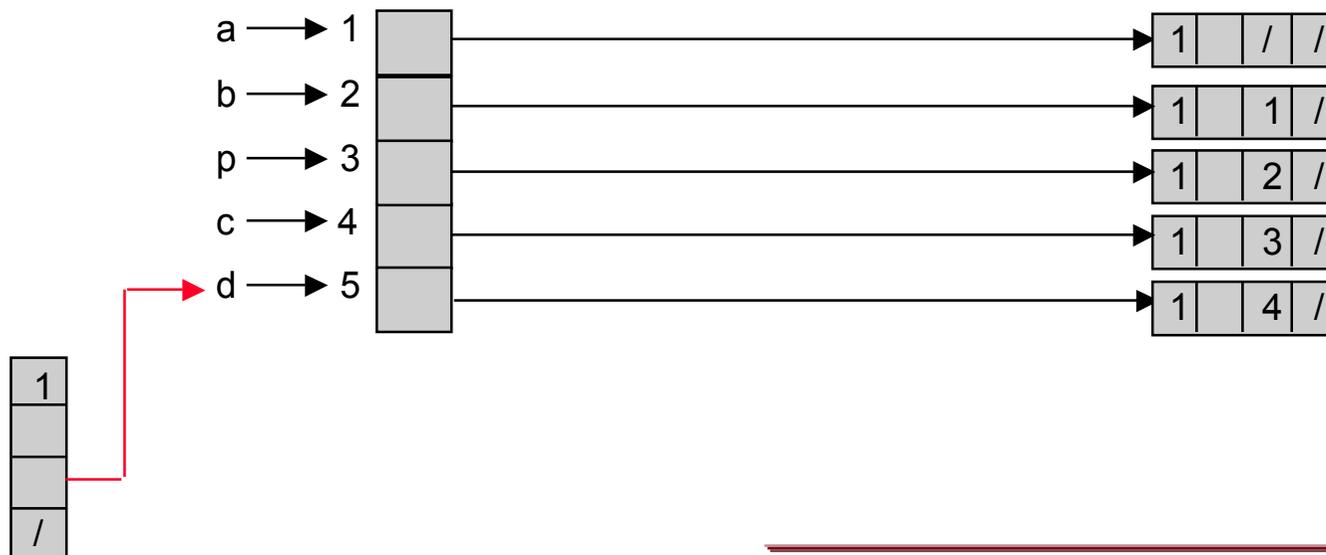
Beispiel



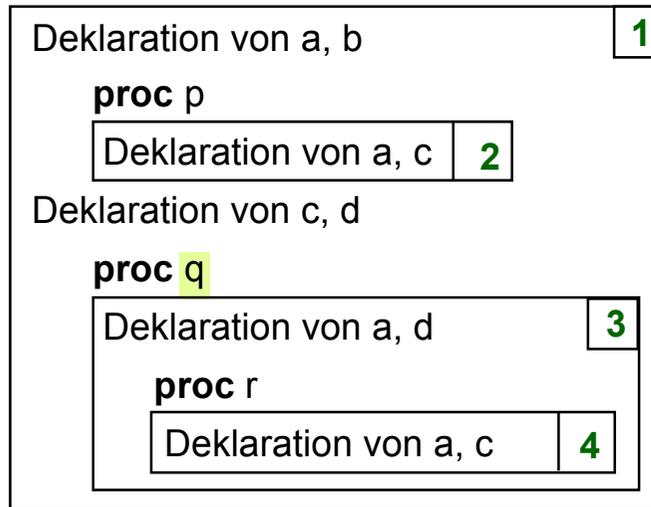
Abarbeitung der Deklarationen

```

create_symb_table
enter_block 1
enter_id(a, ↑(a in 1))
enter_id(b, ↑(b in 1))
enter_id(p, ↑(p in 1))
enter_block 2
enter_id(a, ↑(a in 2))
enter_id(c, ↑(c in 2))
exit_block 2
enter_id(c, ↑(c in 1))
enter_id(d, ↑(d in 1))
    
```



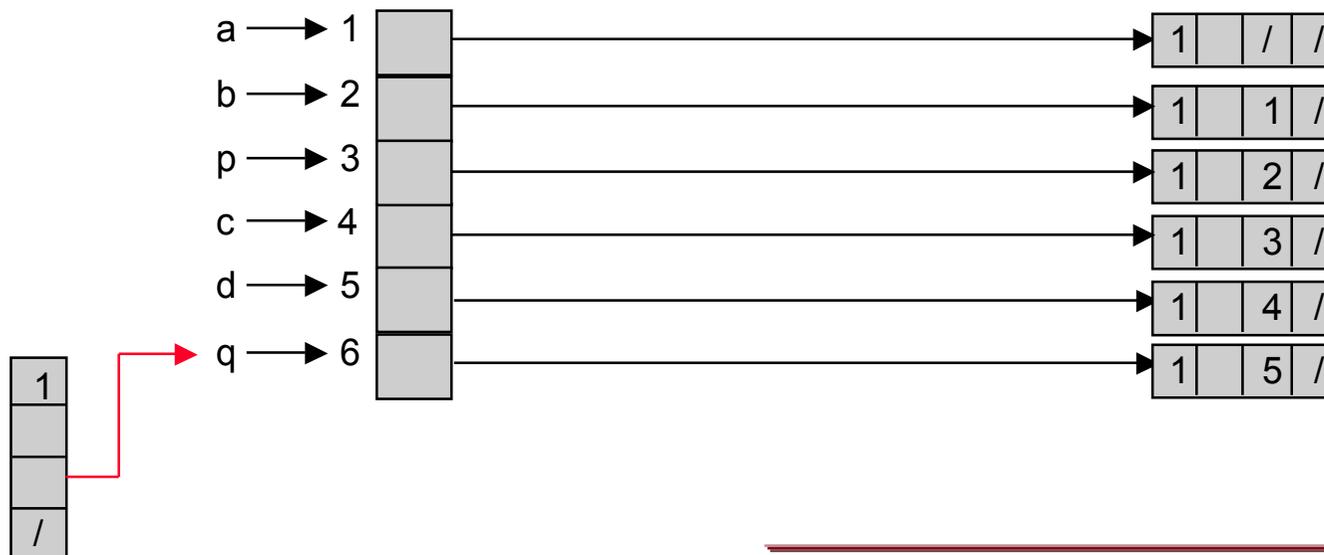
Beispiel



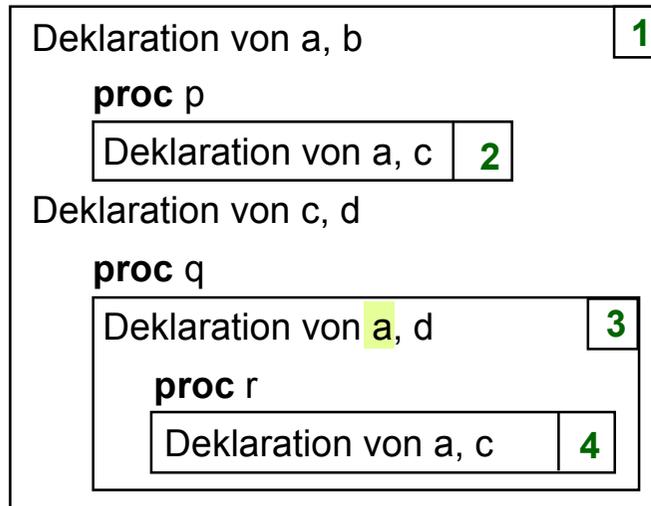
Abarbeitung der Deklarationen

```

create_symb_table
enter_block 1
enter_id(a, ↑(a in 1))
enter_id(b, ↑(b in 1))
enter_id(p, ↑(p in 1))
enter_block 2
enter_id(a, ↑(a in 2))
enter_id(c, ↑(c in 2))
exit_block 2
enter_id(c, ↑(c in 1))
enter_id(d, ↑(d in 1))
enter_id(q, ↑(q in 1))
    
```



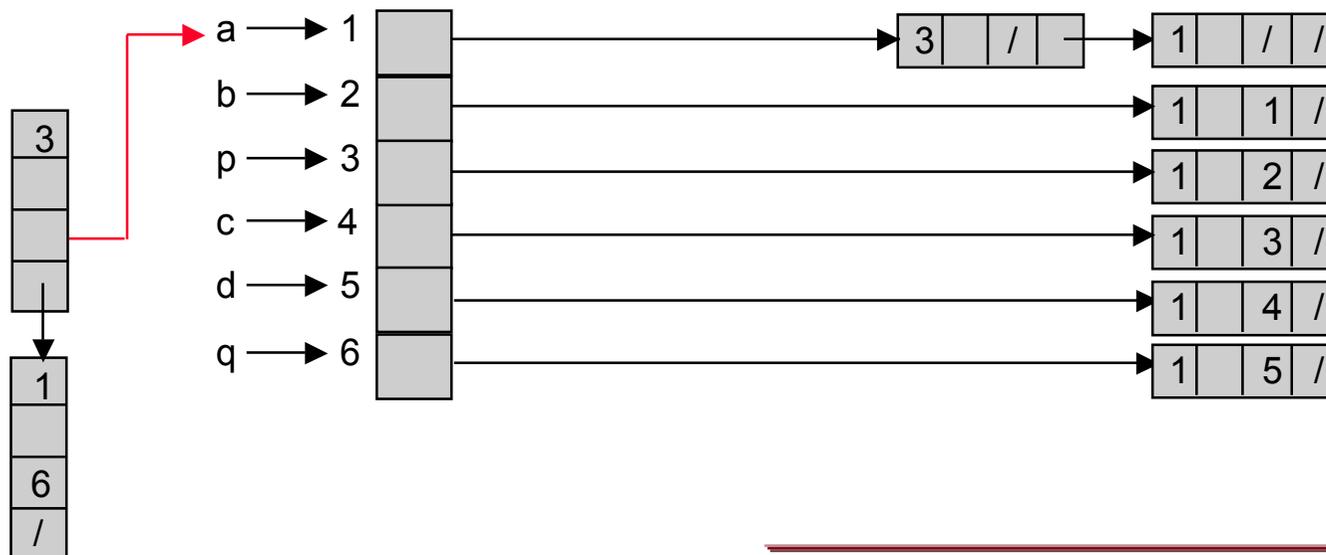
Beispiel



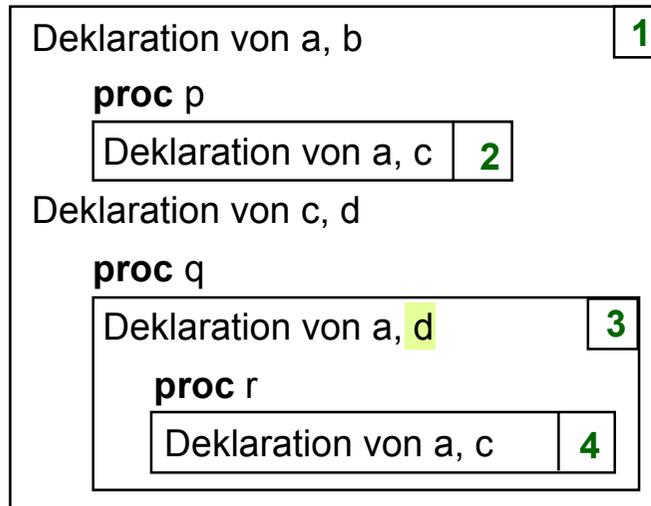
Abarbeitung der Deklarationen

```

create_symb_table
enter_block 1
enter_id(a, ↑(a in 1))
enter_id(b, ↑(b in 1))
enter_id(p, ↑(p in 1))
enter_block 2
enter_id(a, ↑(a in 2))
enter_id(c, ↑(c in 2))
exit_block 2
enter_id(c, ↑(c in 1))
enter_id(d, ↑(d in 1))
enter_id(q, ↑(q in 1))
enter_block 3
enter_id(a, ↑(a in 3))
    
```



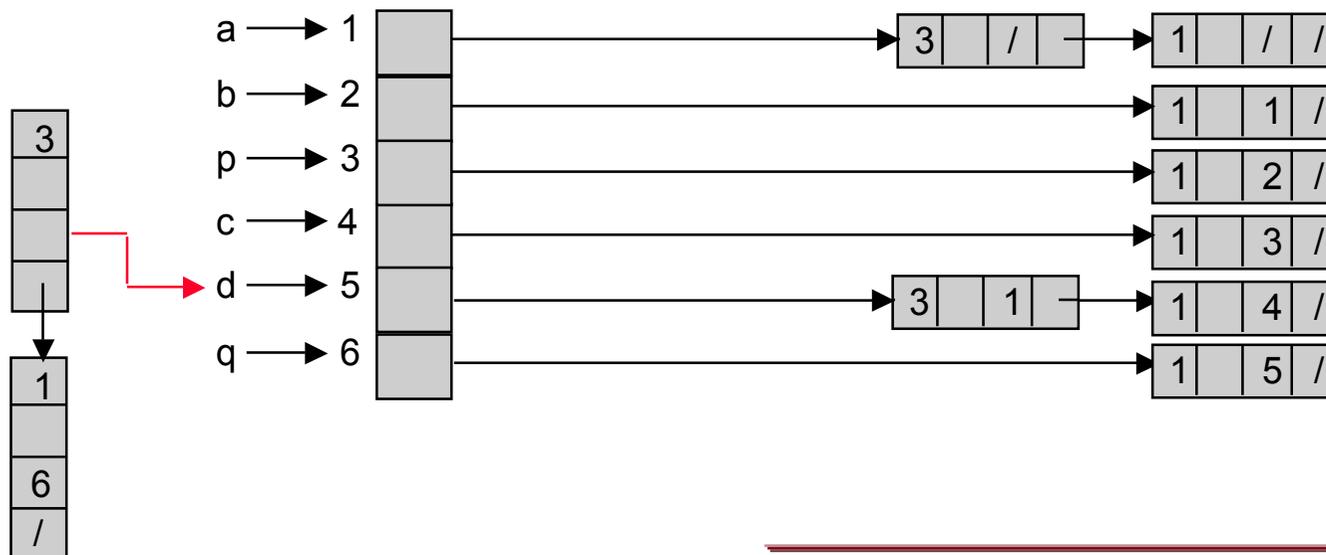
Beispiel



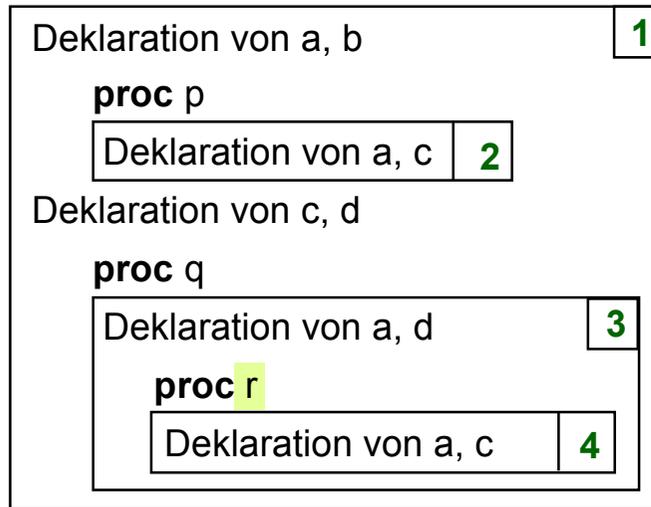
Abarbeitung der Deklarationen

```

create_symb_table
enter_block 1
enter_id(a, ↑(a in 1))
enter_id(b, ↑(b in 1))
enter_id(p, ↑(p in 1))
enter_block 2
enter_id(a, ↑(a in 2))
enter_id(c, ↑(c in 2))
exit_block 2
enter_id(c, ↑(c in 1))
enter_id(d, ↑(d in 1))
enter_id(q, ↑(q in 1))
enter_block 3
enter_id(a, ↑(a in 3))
enter_id(d, ↑(d in 3))
    
```



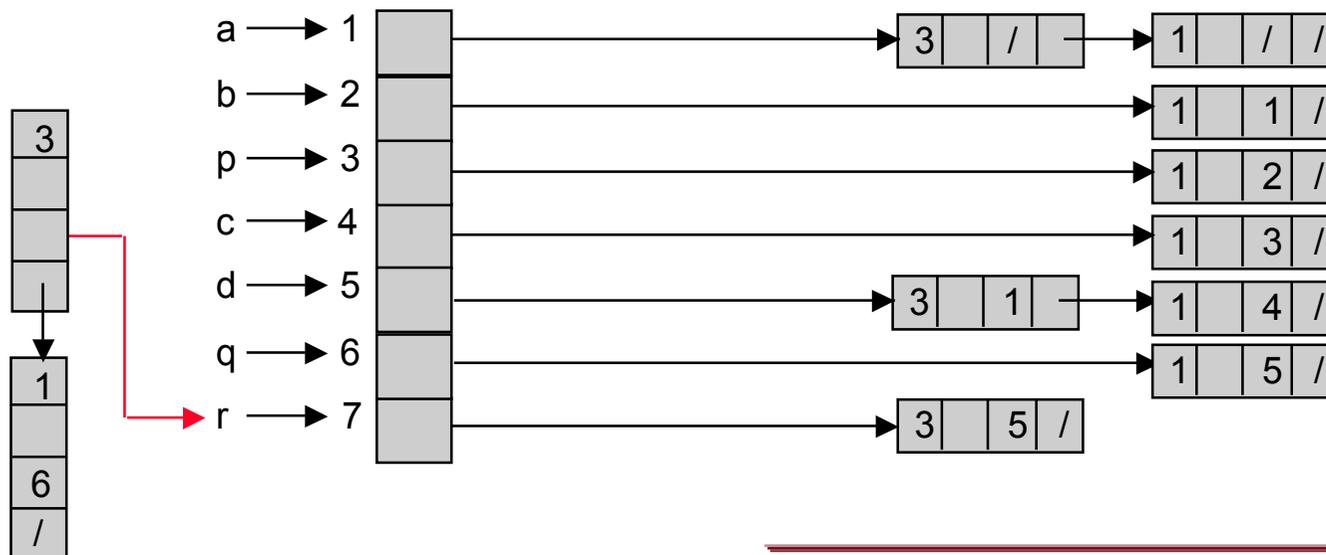
Beispiel



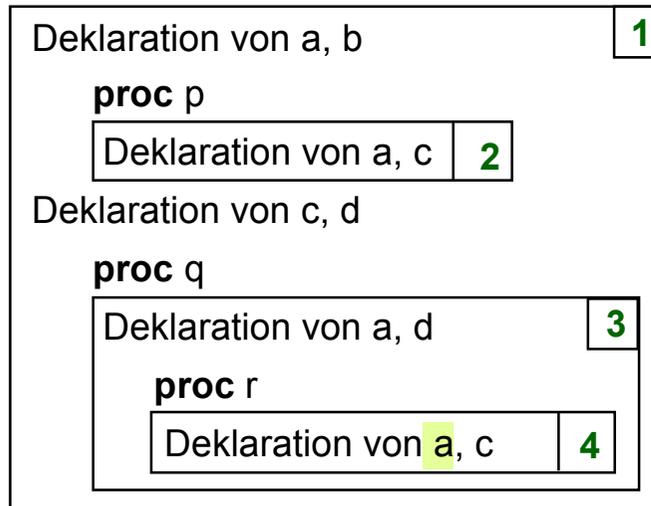
Abarbeitung der Deklarationen

```

create_symb_table
enter_block 1
enter_id(a, ↑(a in 1))
enter_id(b, ↑(b in 1))
enter_id(p, ↑(p in 1))
enter_block 2
enter_id(a, ↑(a in 2))
enter_id(c, ↑(c in 2))
exit_block 2
enter_id(c, ↑(c in 1))
enter_id(d, ↑(d in 1))
enter_id(q, ↑(q in 1))
enter_block 3
enter_id(a, ↑(a in 3))
enter_id(d, ↑(d in 3))
enter_id(r, ↑(r in 3))
    
```



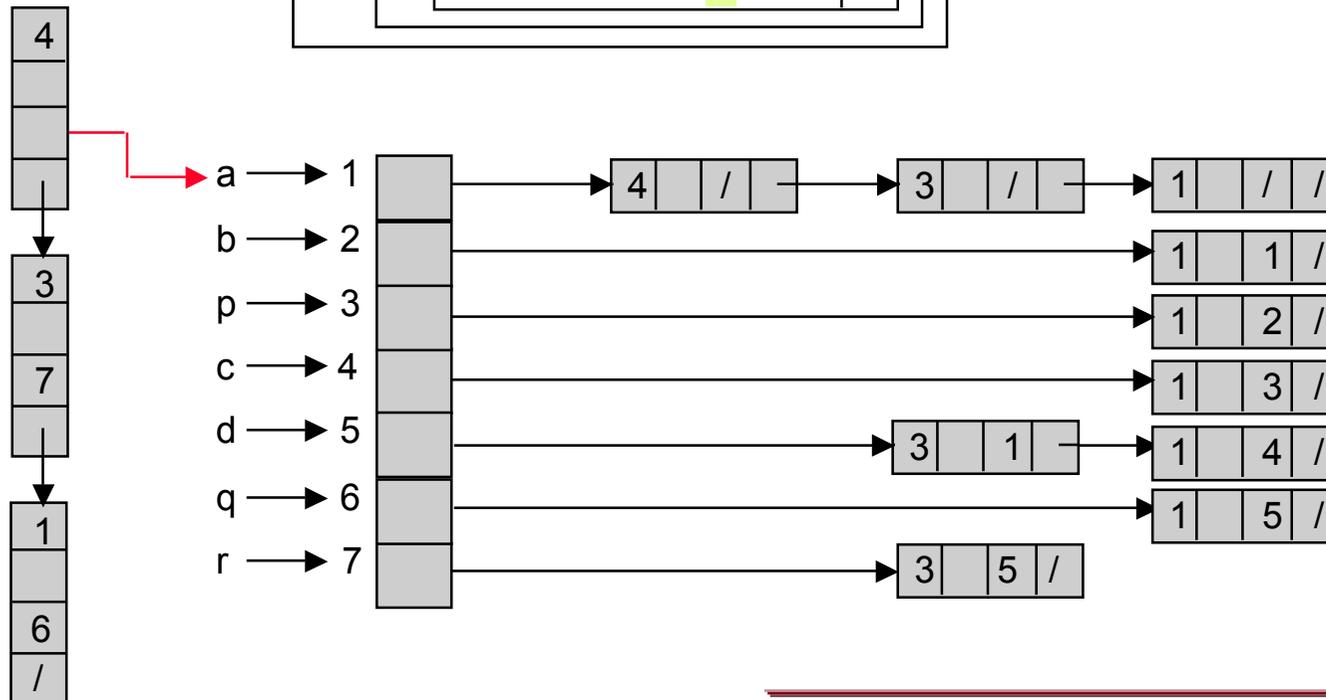
Beispiel



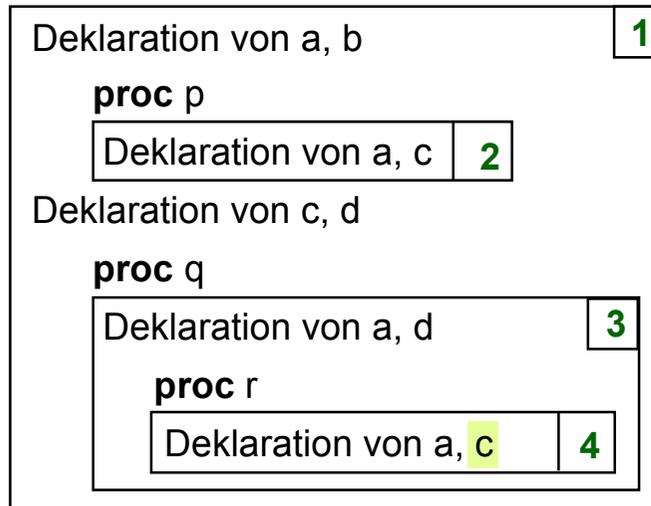
Abarbeitung der Deklarationen

```

create_symb_table
enter_block 1
enter_id(a, ↑(a in 1))
enter_id(b, ↑(b in 1))
enter_id(p, ↑(p in 1))
enter_block 2
enter_id(a, ↑(a in 2))
enter_id(c, ↑(c in 2))
exit_block 2
enter_id(c, ↑(c in 1))
enter_id(d, ↑(d in 1))
enter_id(q, ↑(q in 1))
enter_block 3
enter_id(a, ↑(a in 3))
enter_id(d, ↑(d in 3))
enter_id(r, ↑(r in 3))
enter_block 4
enter_id(a, ↑(a in 4))
    
```



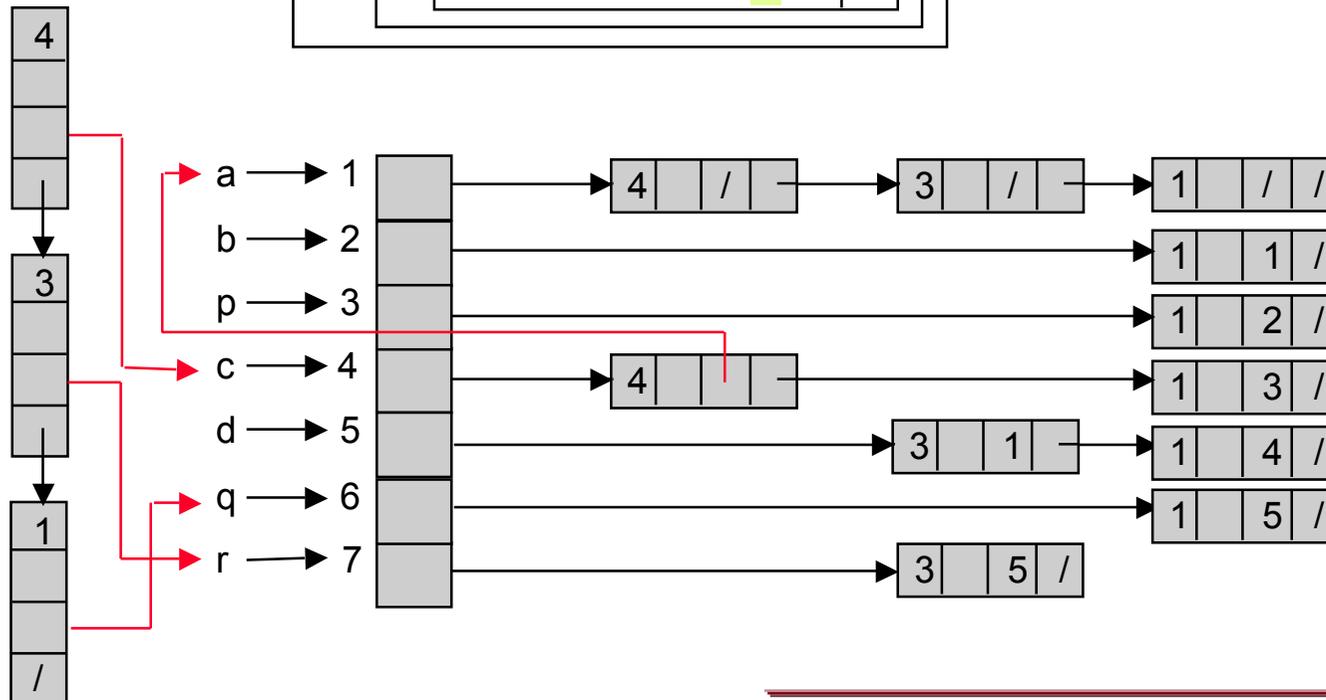
Beispiel



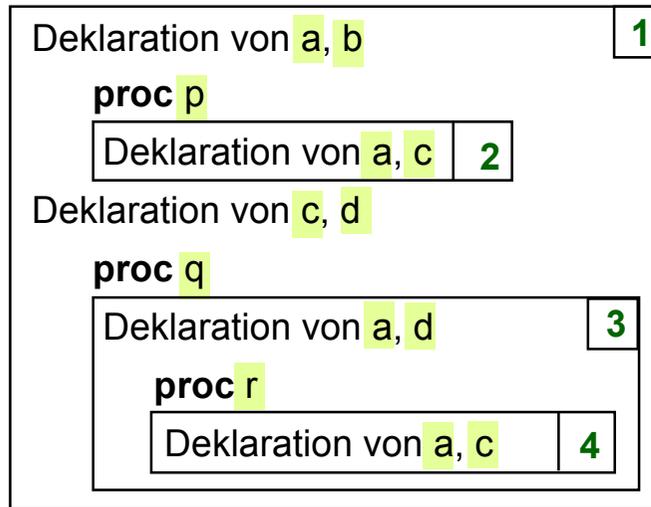
Abarbeitung der Deklarationen

```

create_symb_table
enter_block 1
enter_id(a, ↑(a in 1))
enter_id(b, ↑(b in 1))
enter_id(p, ↑(p in 1))
enter_block 2
enter_id(a, ↑(a in 2))
enter_id(c, ↑(c in 2))
exit_block 2
enter_id(c, ↑(c in 1))
enter_id(d, ↑(d in 1))
enter_id(q, ↑(q in 1))
enter_block 3
enter_id(a, ↑(a in 3))
enter_id(d, ↑(d in 3))
enter_id(r, ↑(r in 3))
enter_block 4
enter_id(a, ↑(a in 4))
enter_id(c, ↑(c in 4))
    
```



Beispiel

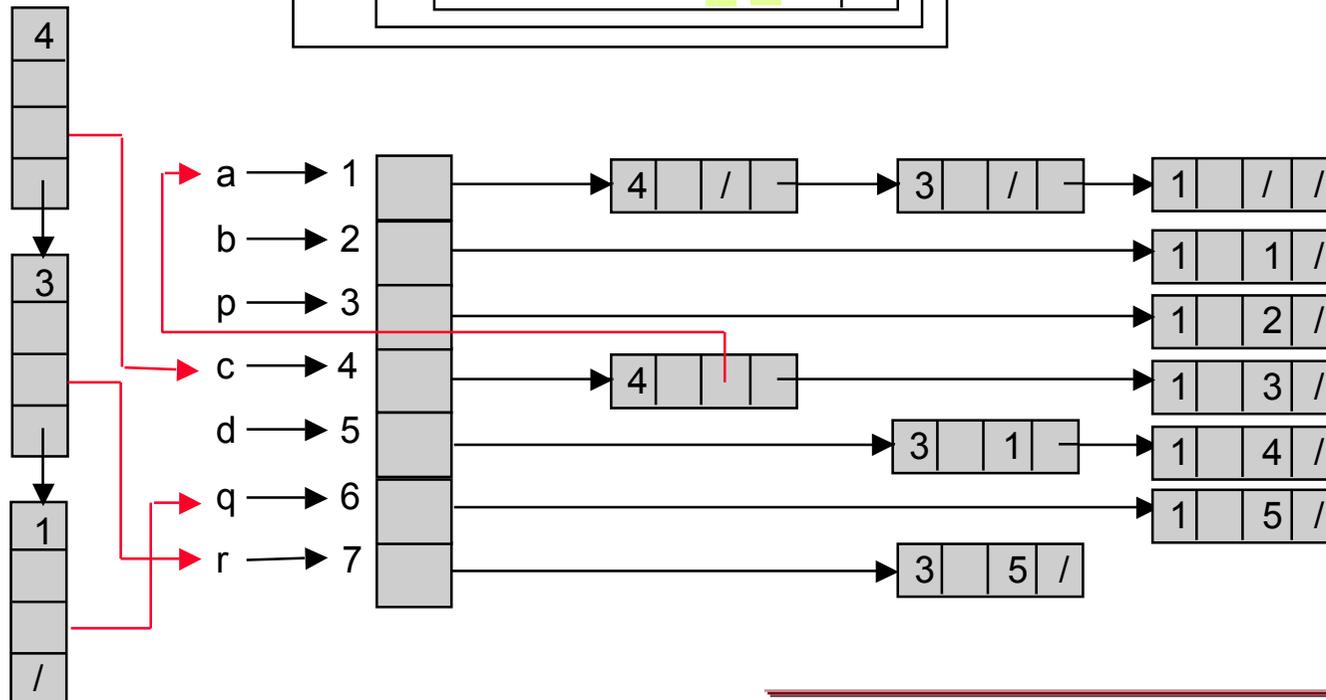


Abarbeitung der Deklarationen

```

create_symb_table
enter_block 1
enter_id(a, ↑(a in 1))
enter_id(b, ↑(b in 1))
enter_id(p, ↑(p in 1))
enter_block 2
enter_id(a, ↑(a in 2))
enter_id(c, ↑(c in 2))
exit_block 2
enter_id(c, ↑(c in 1))
enter_id(d, ↑(d in 1))
enter_id(q, ↑(q in 1))
enter_block 3
enter_id(a, ↑(a in 3))
enter_id(d, ↑(d in 3))
enter_id(r, ↑(r in 3))
enter_block 4
enter_id(a, ↑(a in 4))
enter_id(c, ↑(c in 4))

```



Implementierung der Symboltabelleoperationen

```
proc create_symb_table
  begin kreierte leeren Keller von Blockeinträgen end;
proc enter_block
  begin kellere Eintrag für neuen Block end;
proc exit_block
  begin
    foreach Deklarationseintrag des aktuellen Blocks do lösche Eintrag od;
    entferne Blockeintrag aus dem Keller
  end;
proc enter_id (id: Idno; decl: ↑node);
  begin
    if existiert bereits ein Eintrag für id in diesem Block then error („Doppeldeklaration“) fi;
    kreierte neuen Eintrag mit decl und Nr. des aktuellen Blocks;
    füge diesen Eintrag an die lineare Liste für diesen Block an;
    füge diesen Eintrag (vorne) an die lineare Liste für id an;
  end;
proc search_id (id: Idno) ↑node;
  begin
    if Zeile für id ist leer
      then error („undeklarierter Bezeichner“)
      else return (Wert des decl-Felds aus erstem Eintrag in Zeile id) fi
  end
```

Neuer Listenkopf

Über entsprechende Listenköpfe und blockweise Verkettung

Verweis auf Vorgänger-Deklaration im Block

Verweis auf statischen Vorgänger

D.h. über die 2. Komponente der Listenelemente



Überprüfung der Typkonsistenz

- Bottom-up-Pass über Ausdrucksbäume
 - Konstanten: Typ klar
 - Bezeichner: Typ aus deklarativer Information
 - Operator: Funktionalität in Tabelle nachsehen
- Zusätzlich
 - Bei *Typanpassung* alle möglichen Kombinationen aus Operandentypen prüfen
 - Bei *Überladung* (von Operatoren) richtige Operation auswählen (s.u.)

Überladung von Bezeichnern

- **Überladenes Symbol**
 - Kann in einem Programm mehrere Bedeutungen haben / Z.B. Operationssymbol
- Aufgabe der Typberechnung: jeweils richtige Operation zu überladenem Symbol finden
- Überladung von benutzerdefinierten Symbolen / In Ada: nur für Funkt./Proz., nicht für Variablen
 - Redeklaration verbirgt nur dann äußere Deklaration, wenn Typgleichheit vorliegt
 - Programm ist korrekt, wenn „Typumgebung“ eindeutige Zuordnung von Symbolen zu Operationen erlaubt

Problem

- Konfliktsituationen durch Sichtbarkeitsregeln kombiniert mit Überladung

Beispiel (Ada-Programm)

```

procedure BACH is
  procedure put (x: boolean) is begin null; end;
  procedure put (x: float) is begin null; end;
  procedure put (x: integer) is begin null; end;
  package x is

```

D1 → type **boolean** is (false, true);
 function **f** return boolean; =

D1: boolean, f: *neue Bezeichner*;
 durch use x potentiell sichtbar

```

end x;
package body x is
  function f return boolean is begin null; end;
end x;

```

D2: *Überladung von f*

D2 → function **f** return float is begin null; end;
 use x;

A1: D1 + D2 sichtbar (versch. Parameter-Profile)
 Hier gemeint: D2 (Parametertyp von put)

D3 → A: declare **f**: integer;
 begin

D3: *verdeckt D1 + D2 (Var.-Bezeichner nicht überladbar)*

```

  put (f);

```

A2: *Bezug auf D3*

D4 → B: declare
 function **f** return integer is begin null; end;
 begin

D4: *verdeckt D3 (und transitiv D2);
 D1 bleibt sichtbar (wegen use)*

```

  put (f);
end B;

```

A3: *f bezieht sich auf D4*

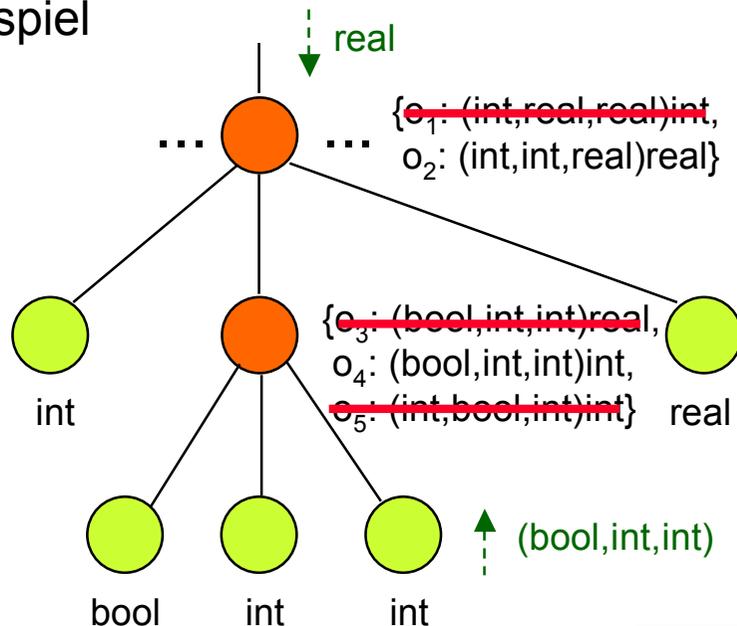
```

end A;
end BACH

```

Auflösung der Überladung (*overload resolution*)

- 4 Läufe über den abstrakten Syntaxbaum
- Dabei zu jedem Knoten
 - `init_ops`: Menge der an jedem Knoten sichtbaren und bezgl. Stelligkeit zulässigen Operatoren
 - `bottom_up_elim`: Elimination der Operatoren, deren *i*-ter Parametertyp kein möglicher Resultattyp des *i*-ten Operanden ist
 - `top_down_elim`: Elimination der Operatoren, deren Resultattyp kein möglicher Parametertyp ist
 - Eindeutigkeitsprüfung
- Beispiel





Auflösungsalgorithmus

```
proc resolve_overloading (root: node, apriori_typ: type);
proc init_ops;
  begin
    foreach k do ops(k) := {op | op ∈ vis(k) and rank(k) = #descs(k)} od;
    ops(root) := {op ∈ ops(root) | res_typ(op) = apriori_typ}
  end;
proc bottom_up_elim (k: node);
  begin
    for i := 1 to #descs(k) do
      bottom_up_elim(k.i);
      ops(k) := {op ∈ ops(k) | par_typ(op, i) ∈ {res_typ(op) | op ∈ ops(k.i)}} od
    end;
proc top_down_elim (k: node);
  begin
    for i := 1 to #descs(k) do
      ops(k.i) := {op ∈ ops(k.i) | res_typ(op, i) ∈ {par_typ(op, i) | op ∈ ops(k)}};
      top_down_elim(k.i) od;
    end;
  begin
    init_ops;
    bottom_up_elim(root);
    top_down_elim(root);
    prüfe, ob jetzt alle ops-Mengen einelementig sind; sonst Fehlermeldung
  end
end
```

symb(k): Symbol, mit dem k markiert ist
vis(k): an k sichtbare Definitionen von symb(k)
#descs(k): Anzahl Kinder von k

Resultattyp von op

Typ des i-ten Parameters von op

Nur die Operatoren, deren i-ter Parameter ein möglicher Resultattyp des i-ten Operanden ist

Nur die Operatoren, deren Resultattyp zu einem Typ des Zugehörigen Parameters passt



Typäquivalenz

- Durch Unifikation von Typausdrücken

(Parametrischer) Polymorphismus

- Verwendung von Typvariablen bzw. Typausdrücken
- Definition von Funktionen, die für eine Menge von Operanden- und Resultattypen i.w. dasselbe tun
- Wird verwendet in
 - Funktionalen Sprachen
 - Ada: generische Pakete

*Einzelheiten dazu in der Vorlesung
„Funktionale Sprachen“*

Behandlung (des Polymorphismus) durch Typinferenzalgorithmus

- Berechnet für jede Anwendung einer polymorph getypten Funktion den richtigen Typ
- Stellt fest, ob all diese so berechneten Typen ein gemeinsames Typschema haben (das dann der Typ der Funktion ist)