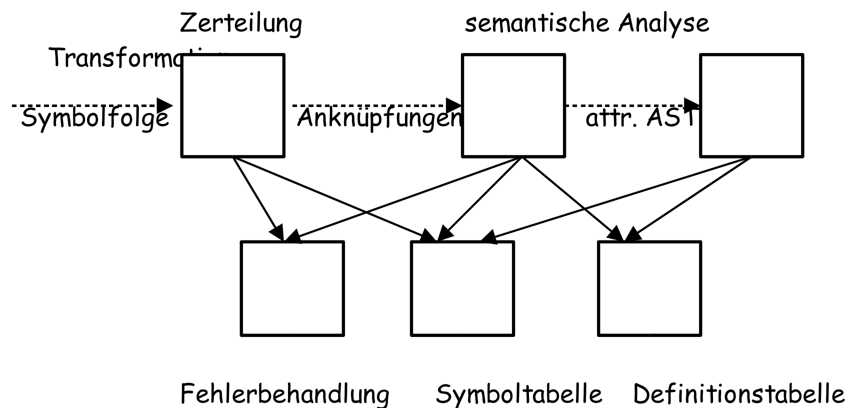
5. Kapitel

Semantische Analyse
Teil 2

Kapitel 5: Semantische Analyse

0. Einbettung
1. Namensanalyse
    1.1 Implementierung
2. Typanalyse und Operatoridentifikation
    2.1 Typanalyse
    2.2 Zusammenhänge
    2.3 Implementierung / Beispiel

5. Semantische Analyse



5. Semantische Analyse: Aufgaben

formal:
- statische Semantik berechnen: die in der Syntaxanalyse versäumten Aufgaben nachholen
- Konsistenzprüfung entsprechend Sprachdefinition

praktisch:
- Namensanalyse: Bedeutung der Bezeichner feststellen
- Typanalyse: Typen aller Ausdrücke bestimmen
- Operatoridentifikation: Bedeutung der Operatoren bestimmen
- Konsistenzprüfung
- sprachabhängige Sonderaufgaben

Schwierigkeiten:
- Aufgaben ineinander verschränkt
- komplexe Datenstrukturen für Namensanalyse (Gültigkeitsbereichsdefinitionen)
- umfangreiche Suchaufgaben: Zeitaufwand reduzieren
- Definitionstabelle (DB: Eigenschaften von Bezeichnern, ...) auch später nötig

Kapitel 5: Semantische Analyse

0. Einbettung
1. Namensanalyse
    1.1 Implementierung
2. Typanalyse und Operatoridentifikation
    2.1 Typanalyse
    2.2 Zusammenhänge
    2.3 Implementierung / Beispiel

5.1 Namensanalyse

Unterscheide

- Bezeichnerdefinition (*defining occurrence*): Vereinbarung, Parameterspezifikation, Markendefinition, vordefiniert, ...
    - vordefiniert: wie ein das Gesamtprogramm umschließender Block
- Bezeichneranwendung (*applied occurrence*): Benutzung als Variable, Konstante, Parameter, Verbund-/Objekt-/Modul-attribut (Feld), Prozedurname im Aufruf, Typ, Sprungziel, ...
    - Sonderfälle:
        - Schlüsselwortparameter *p*(*filename* = "abc", *condition* = ...)
        - Schleifenmarken: *m*: **loop** ... **loop** ...**exit** *m*; ... **end**; ... **end**
        - unvollständige Spezifikation von Feldern: *a.c* statt *a.b.c* (Cobol, PL/1), with-Anweisung (Pascal, Modula)
        - implizite Definition (Fortran 77)

Zuordnung Anwendung → Definition abhängig von Gültigkeitsbereichsregeln, syntaktischer Position: unterschiedliche Namensräume

5.1 Namensräume: Grobklassifikation

1. Globale Definitionen
2. Modul-/Klassen-/Objekt-/Verbund-Definitionen
3. Lokale Definitionen (lokale Variable, Parameter) entsprechend Blockschachtelung – Konturmodell
4. Sonderfälle

Klassifikation steuert Gebrauch des Attributs Umgebung (umg).

5.1 Blockschachtelung: Grundschema

```
1. rule block ::= deklarationen ; anweisungen .
   attribution
   anweisungen.umg := append(deklarationen.umg, block.umg)
2. rule  deklarationen ::= deklarationen ´;´ deklaration.
   attribution
   deklarationen[1].umg := append(deklarationen[2].
  umg,deklaration.umg)
3. rule deklaration ::= bezeichner ´:´ typ .
   attribution
   deklaration.umg := new Umg(bezeichner.symbol, typ.deftab,...)
4. rule anweisungen ::= anweisungen ´;´ anweisung .
   attribution
   anweisungen[2].umg:= anweisungen[1].umg;
   anweisung.umg:= anweisungen[1].umg
5. rule anweisung ::= . . . Variable . . .
   attribution
   variable.deftab := anweisung.umg.search(variable.symbol)
```

nicht behandelt: woher erhält man die Eigenschaften von typ ?

5.1 Umgebungsattribut *umg*

ererbtes Attribut in Regeln 1, 4, 5

synthetisiertes Attribut in Regel 2 und 3

Behandlung von Bezeichneranwendungen, z.B. Initialisierungen, in Vereinbarungen?

rule deklaration ::= bezeichner ´:´ typ ´:=´... variable ...

Lösung: Unterscheide umg_ein – umg_aus

Bezeichneranwendungen werden mit umg_ein identifiziert

umg_ein umfaßt

- *alle* Definitionen des Blocks und seiner Umgebung, oder
- *nur* die vorangehenden Definitionen, oder
- Mischungen aus beidem

5.1 Blockschachtelung: nur vorangehende Vereinbarungen

```
1. rule block ::= deklarationen ; anweisungen .
   attribution
   anweisungen.umg := deklarationen.umg_aus;
   deklarationen.umg_ein := block.umg
2. rule deklarationen ::= deklarationen ´;´ deklaration.
   attribution
   deklarationen[1].umg_aus:= deklaration.umg_aus;
   deklarationen[2].umg_ein :=deklarationen[1].umg_ein;
   deklaration.umg_ein := deklarationen[2].umg_aus;
3. rule deklaration ::= bezeichner ´:´ typ ´:=´ variable ´;´.
   attribution
   typ.deftab := deklaration.umg_ein.search(typ.symbol);
   variable.deftab:= deklaration.umg_ein.search(variable.symbol);
   deklaration.umg_aus :=
     append(new Umg(bezeichner.symbol,typ.deftab,...),
            deklaration.umg_ein);
...
```

Schema ist $LAG(1)$ und OAG

5.1 Attribut „durchschleifen“ umg_ein - umg_aus

umg_ein ist ererbtes Attribut in Regeln 1 und 2

umg_aus ist synthetisiertes Attribut in Regeln 2 und 3

Gleiche Technik in Anweisungen, wenn dort Bezeichnerdefinitionen erlaubt, z.B. bei impliziten Vereinbarungen in Fortran

Wenn alle Definitionen des Blocks und seiner Umgebung erlaubt (Verwendung vor Vereinbarung):

Zweifacher Durchlauf:

(a) Definitionen aufsammeln in umg_ein

(b) Definitionen verwenden aus umg_aus

Schema ist $LAG(2)$ und OAG

5.1 Blockschachtelung: Verwendung vor Vereinbarung

```
1. rule block ::= deklarationen ; anweisungen .
   attribution
   anweisungen.umg := deklarationen.umg_aus;
   deklarationen.umg_aus:=append(block.umg,deklarationen.umg_ein)
2. rule deklarationen ::= deklarationen ´;´ deklaration.
   attribution
   deklarationen[1].umg_ein := append(deklarationen[2].umg_ein,
   deklaration.umg_ein);
   deklarationen[2].umg_aus := deklarationen[1].umg_aus;
   deklaration.umg_aus := deklarationen[1].umg_aus
3. rule deklaration ::= bezeichner ´:´ typ ´:=´ variable ´;´.
   attribution
   typ.deftab := deklaration.umg_aus.search(typ.symbol);
   variable.deftab:=
             deklaration.umg_aus.search(variable.symbol);
   deklaration.umg_ein :=
             new Umg(bezeichner.symbol,typ.deftab,...);
. . .
```

5.1 Pascal, Modula-2

Beispiel:

```
type t = ...
var a: integer;
procedure p;
  procedure q; begin a := true end;
  b: t;
  a: Boolean;
  type t' = record a: t; ... end;
  type t =  ref t';
  ...
begin (* p *)
...
end; (* p *)
```

Definition von
1. t
2. t
3. a
??

5.1 Pascal, Modula-2: häufigste Lösung

Beispiel:

```
type t = ...
var a: integer;
procedure p;
  procedure q; begin a := true end;
  b: t;
  a: Boolean;
  type t' = record a: t; ... end;
  type t =  ref t';
  ...
begin (* p *)
...
end; (* p *)
```

Definition von
1. t
2. t
3. a
??

5.1 Pascal, Modula-2: korrekte Lösung

Beispiel:

```
type t = ...
var a: integer;
procedure p;
  procedure q; begin a := true end;
  b: t;
  a: Boolean;
  type t' = record a: t; ... end;
  type t =  ref t';
  ...
begin (* p *)
...
end; (* p *)
```

Definition von
1. t
2. t
3. a
??

5.1 Verbundfelder in Pascal

- Zugriff auf Feld a: $x.a$, **with** x **do begin** ... a ... **end**

  Verfahren für $x.a$:

  1. (Verbund-)Typ t von x bestimmen
  2. Namensraum t, alle Felder des Verbunds, öffnen
  3. In diesem Namensraum (Umgebungsattribut) a suchen
- Namensräume sind also nicht nur Prozedurrümpfe und Blöcke, sondern auch Verbundtypen
- Bei Identifikation von $x.a$ wird nur der Verbundtyp als Umgebung verwandt: keine Vererbung des äußeren Umgebungsattributs wie bei geschachtelten Blöcken
- qualifizierter Zugriff $x.a$ auf Attribute in Moduln, Klassen, Objekten, ... analog

5.1 Die with Anweisung

- Die with Anweisung **with** x **do begin** ... a ... **end** öffnet den Verbundtyp t von x als Namensraum zusätzlich zum Umgebungsattributs des Kontexts
  - Zusatzproblem bei **with** x,y **do begin** ... a ... **end**: Die Typen von x,y könnten beide a definieren, die letzte Definition zählt.
  - Reihenfolge der Suche: zuerst in den Verbundtypen (letzter zuerst), dann im Umgebungsattribut des Kontexts
- Analog im Rumpf von Moduln: zuerst nach lokalen Variablen suchen, dann nach Attributen des Moduls
- Analog im Rumpf von Klassen: zuerst nach lokalen Variablen suchen, dann nach Attributen der Klasse, dann nach Attributen von Oberklassen

5.1 Teilweise Qualifikation

In COBOL, PL/1: Wenn Verbund x Unterverbunde p,... enthält, die ein Feld a definieren, kann statt $x.p.a$ geschrieben werden:

- a, wenn a global eindeutig: es gibt im Programm nur diese Definition von a
- $x.a$, wenn a relativ zu x eindeutig: es gibt im Verbund x nur diese Definition von a
- $p.a$, wenn $p.a$ global eindeutig: es gibt keinen weiteren Verbund p mit einer Definition von a

Allgemeine Regeln: Ein qualifizierter Name ist entweder

- eine vollständige Qualifikation oder
- identifiziert global genau eine Definition

5.1 Teilweise Qualifikation am Beispiel

Schematisches PL/1-Programm:

```
a:     procedure;
declare
1 w,
... ;
   b:  procedure;
   declare
   p,
   1 q,
   2 r,
   3 z,
   2 x,
   3 y,
   3 z,
   3 q;
```

```
y = r.z;          /* q.x.y aus b, q.r.z aus b */
w = q, by name; /* w aus a, äußeres q aus b */
   c: procedure
      declare y,
              1 r,
              2 z;
     z = q.y;        /* r.z aus c, q.x.y aus b */
     x = r, by name;/* q.x aus b, r aus c */
   end c;
  end b;
end a;
```


5.1 Ergebnis der Namensanalyse

- ein Eintrag für jede Definition in der Definitionstabelle
- Verweis auf diesen Eintrag für jede Anwendung

- Definitionstabelle: die zentrale Datenbank des Übersetzers, unstrukturierte Menge von Definitionseinträgen

Kapitel 5: Semantische Analyse

0. Einbettung
1. Namensanalyse
   1.1 Implementierung
2. Typanalyse und Operatoridentifikation
   2.1 Typanalyse
   2.2 Zusammenhänge
   2.3 Implementierung / Beispiel

5.1.1 Implementierung der Namensanalyse

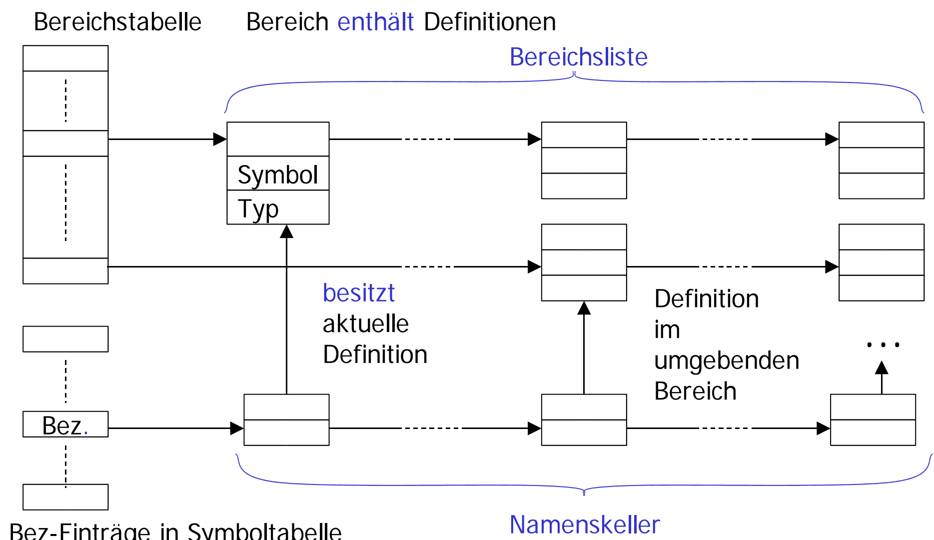
- Hauptaufgabe: Effiziente Implementierung von umg: Namenstabelle
  - Suchen in der Umgebung vermeiden, Ziel $O(1)$
  - Namenstabelle als zentrale Datenstruktur außerhalb des Strukturbaums speichern
    - Aufbau unabhängig von Besuchssequenzen
    - Bei Eintritt/Verlassen eines Namensraums ändert sich der gültige Teil der Tabelle

5.1.1 Namenstabelle

Ansatz:

- Tabelle besteht aus je einer Liste von Definitionen für jeden Bereich
  - Bereich: lokaler Namensraum (Gültigkeitsbereich), globale Namen ausgeschlossen
- Für jede Bezeichneranwendung Keller von Verweisen auf zulässige Definitionen
  - besitzt-Relation: Anwendung besitzt potentielle Definitionen
- erste Definition im Keller ist die richtige
  - von sprachabhängigen Ausnahmen abgesehen, z.B.
    - bei mehreren Definitionen einer Prozedur p mit unterschiedlicher Parameterzahl/Signatur: die erste passende Definition

5.1.1 Namenstabelle



5.1.1 *ADT* Namenstabelle

```
abstract class nameTable {

  private Range currentRange;

  public Range newRange ();
  public void enterRange (Range r);
  public void leaveRange ();       // verläßt currentRange
  public Definition currentDefinition(Symbol s);
  public Definition definitionInRange(Symbol s,Range r);
}
```

5.1.1 Namenstabelle II

Erkenntnis: Namenskeller aus der Symboltabelle ansteuern

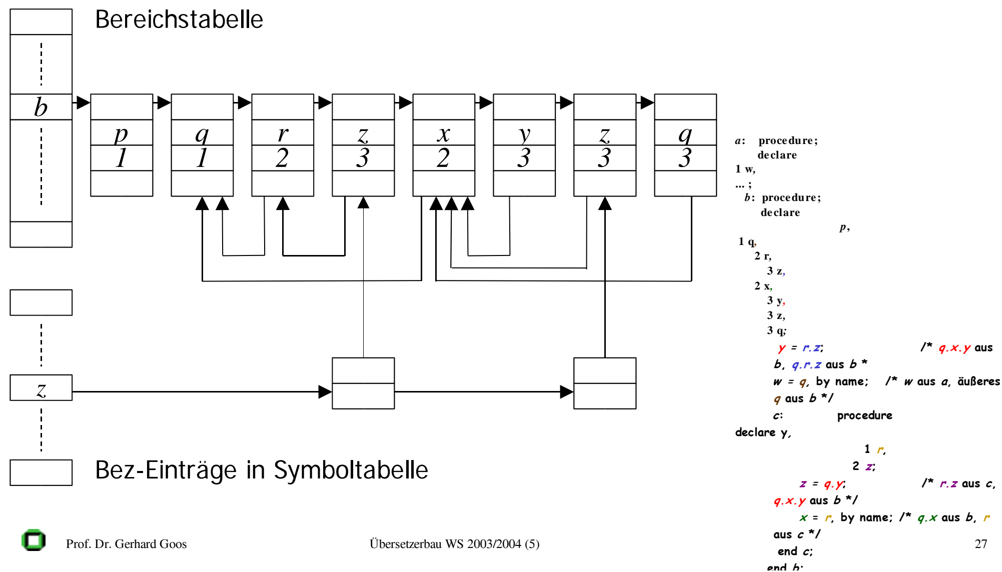
Namenskeller bei Eintritt/Verlassen eines Namensraums ändern

Suchaufwand konstant, zusätzlicher Aufwand bei Eintritt/Verlassen eines Namensraums

Behandlung von Sonderfällen:

- Pascal/Modula-**with**, auch geschachtelt, schon berücksichtigt:
  - Bereichsliste eines Verbundtyps (einer Klasse) enthält Einträge, die auf mehrere Verbundvariable (Objekte) zutreffen können
  - Daher Unterscheidung der aktuellen Felder im Namenskeller von den Einträgen im Typ
- Erweiterung für teilweise Qualifikation (Voraussetzung: Vereinbarung Verbundvariable/Objekt = Typdefinition)
  - alle Verbundfelder auch von Unterverbunden flach in einer Bereichsliste, aber Schachtelungstiefe vermerken und Rückverweis auf Vater eintragen: nächste Folie
  - Testalgorithmus übernächste Folie

5.1.1 Namenstabelle: teilweise Qualifikation



5.1.1 Test teilweise Qualifikation

```
/* Teste, ob p durch qualName identifizierter Eintrag ist
   Test für alle p im Namenskeller von ausgehend
   qualName[qualName.length-1] durchführen, dann entscheiden
*/
boolean test(symbol[] qualName, Bereichseintrag p) {

   int i = qualName.length - 1;

   while (p != null && i <= p.tiefe) {
      if (qualName[i] == p.symbol) {
         i--; if (i<0) return true;
      }
      p = p.umfassenderVerbund;
   }

   return false;
}
```

5.1.1 Vordefinierte Namen

1. Gewöhnliche Definitionen für alle Programme gültig: Vorbesetzung Namens- und Definitionstabelle
2. Funktionen generisch für verschiedene Parametertypen definiert:
   - Wenn innerhalb der Sprache definierbar (Sprache kennt Überladen, Generizität oder Polymorphie), dann Fall 1
   - sonst Sonderbehandlung: Übersetzer führt generische Parameter- und Ergebnistypen, die es in der Sprache nicht gibt, intern ein, dann Fall 1
3. Funktionen mit variabler Parameterzahl:
   - Wenn innerhalb der Sprache definierbar, dann Fall 1 (selten außer in interpretierten Sprachen, z.B. Kommandosprachen)
   - sonst Sonderbehandlung:
     - Übersetzer führt Funktionen mit variabler Parameterzahl intern ein (oft für Ein-/Ausgabe) oder
     - Baumtransformation in Aufrufe von Funktionen mit fester Parameterzahl (häufig sind diese Funktionen nicht explizit in der Sprache zugänglich) oder explizite Codierung (Makrosubstitution, z.B. maximum-Bildung in Fortran)

Vorsicht: Sonderbehandelte Funktionen nicht als formale Prozeduren zulässig! Das steht oft nicht im Sprachbericht!

5.1.1 Zusammenfassung Namensanalyse

Namensanalyse liefert für alle Bezeichneranwendungen einen Verweis *deftab* auf einen Eintrag in der Definitionstabelle

Der Verweis ist persistent: Attribut wird in der Transformationsphase noch benötigt

Abhängigkeit der Namensanalyse von der Typanalyse:

- bei qualifizierten Namen $x.a$, with-Anweisungen, usw. muß Typ des Qualifikators x bekannt sein
- bei Vererbung/Generizität in oo-Sprachen müssen die zulässigen Attribute/Funktionen der Oberklassen bzw. der Typargumente bekannt sein
- bei Identifikation von Funktionen abhängig von der Signatur, z.B. bei überladenen Funktionen, müssen die Argumenttypen bekannt sein

Hinweis:

Operatoridentifikation ist signaturabhängige Namensanalyse von Funktionen!

Kapitel 5: Semantische Analyse

0. Einbettung
1. Namensanalyse
   1.1 Implementierung
2. Typanalyse und Operatoridentifikation
   2.1 Typanalyse
   2.2 Zusammenhänge
   2.3 Implementierung / Beispiel

5.2.1 Typanalyse

Typ: Kennzeichnung von Objekten bezüglich zulässiger Wertemenge und zulässigen Operationen

- einschließlich impliziter Typanpassungen, z.B. $int \rightarrow real$, dereferenzieren, deprozedurieren (parameterlose Funktion → Wert), vereinigen (Wert → Vereinigungstyp)

Aufgabe der Typanalyse:

- Typen aller Namen, Operanden, Ausdrucksergebnisse bestimmen
  - notwendig für Namensanalyse und Operatoridentifikation
  - notwendig zur Bestimmung von Typanpassungen
  - notwendig für Konsistenzprüfung (Programmiersicherheit)

Unterscheide Sprachen:

- stark typisiert (statisch oder dynamisch, Pascal, Modula, Ada, Sather, Java, C#, ..., fast immer mit Einschränkungen)
- schwach typisiert (C, C++,...)
- typfrei (Maschinensprachen, ...: Operationen typisiert, Objekte nur durch Umfang und Ausrichtung im Speicher gekennzeichnet)

bei funktionalen Sprachen Typanalyse durch Typinferenz!

5.2.1 Uniformer Zugriff

Uniformer Zugriff (*uniform referents*): (Lesender) Zugriff auf Daten soll unabhängig von der Art der Implementierung notiert werden.

- Ziel: Trennung von Anwendung und Implementierung

Daher:

- Variablenzugriff v und Zugriff p mit parameterlosem Aufruf gleich schreiben, also nicht $p()$.
- Reihungszugriff $a[i,j]$ und Prozeduraufruf $p(i,j)$ mit Parametern gleich schreiben (wie in Fortran)

Typabgleich (balancing) und -anpassung im Übersetzer ermöglichen dies.

Beispiel: Sather-K und C#, sowie die meisten akademischen OO-Sprachen benutzen das.

5.2.1 Typäquivalenz

Unterscheide:

- Namensgleichheit: zwei Typen t, t' sind gleich, wenn sie durch die gleiche Typdefinition definiert werden.
- Strukturgleichheit: Zwei Typen t, t' sind gleich, wenn sie durch den gleichen Typkonstruktor mit den gleichen Argumenten (Bezeichner und Typ von Verbundfeldern, Anzahl und Typ der Indexgrenzen, Typ der Reihungselemente, usw.) erzeugt werden können
  - Typen als Terme auffassen, Terme vergleichen
  - Vorsicht: Typen können rekursiv sein, die Terme sind dann unendlich!
- Verfahren zur Überprüfung Strukturgleichheit in der Übung

5.2.1 Typattribute

bereits bekannt:

- *a priori* Typ: synthetisiertes Attribut *vor*
- *a posteriori* Typ: ererbtes Attribut *nach*
- dazwischen Typanpassung

Typ von Namen, Objekten in Definitionstabelle eingetragen

- wird mindestens zur Speicherzuteilung gebraucht

ansonsten Typattribute nach Namensanalyse, Operatoridentifikation, Feststellung Typanpassung, Konsistenzprüfung überflüssig außer:

- bei dynamischer Typprüfung (und dynamischer Operatoridentifikation)
- bei Verwendung von Vereinigungs- und polymorphen Typen

Kapitel 5: Semantische Analyse

0. Einbettung
1. Namensanalyse
   1.1 Implementierung
2. Typanalyse und Operatoridentifikation
   2.1 Typanalyse
   2.2 Zusammenhänge
   2.3 Implementierung / Beispiel

5.2.2 Zusammenspiel Typanalyse - Operatoridentifikation

1. *a priori* Typ der Operanden von $op1\ \tau\ op2$ übernehmen
2. mögliche Definitionen von τ feststellen (Menge)
3. Auswahl unter diesen Definitionen bzw. Fehlermeldung, wenn keine oder mehr als eine Definition zulässig (Operator identifizieren, liefert Operation), a priori Typ des Ergebnisses bestimmen
4. *a posteriori* Typen der Operanden $op1$, $op2$ berechnen
5. Typanpassung *a priori* → *a posteriori* Typ bestimmen und als Attribut merken

Beispiel: In $a[e]$ durch Verwendung klar: a ist Reihung, e ist Ausdruck des Typs der Grenzen von a

5.2.2 Eindeutigkeit von Operator- und Funktionsdefinitionen

nachgeprüft bei Eintrag in die Definitionstabelle (Algol´68)

- ineffizient, da Eindeutigkeit für alle legalen Programme zu einer Menge von Definitionen garantiert werden muß.

nachgeprüft beim Finden einer Definition in der Definitionstabelle (alle anderen Programmiersprachen)

- effizient, da Eindeutigkeit nur für konkretes Programm und konkreten Kontext garantiert werden muß.

5.2.2 Typanalyse im Ausdrucksbaum

Unterscheide

A. Operatoridentifikation hängt nur von den Operandentypen ab (alle anderen Sprachen)

B. Operatoridentifikation hängt auch vom verlangten Ergebnistyp ab (linke Seite einer Zuweisung, Ada)

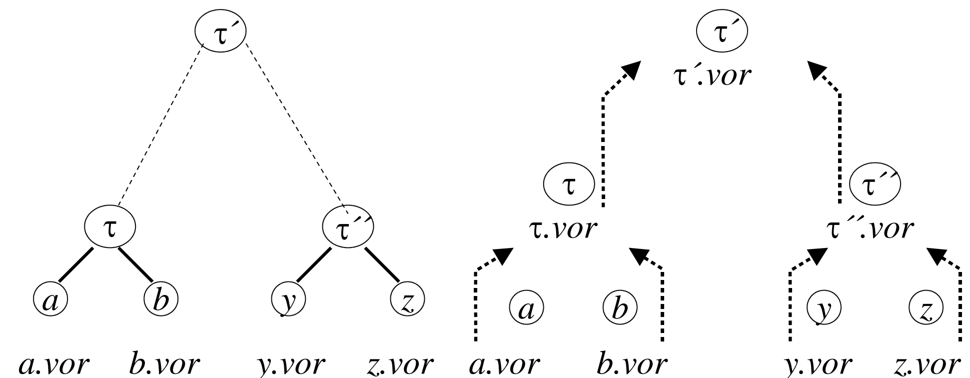
Fall A:

- Schritte 1-5 des Algorithmus von unten nach oben im Baum durchführen (bei Rückkehr aus Tiefensuche)
  - formal $LAG(2)$, da ererbtes Attribut zu spät berechnet

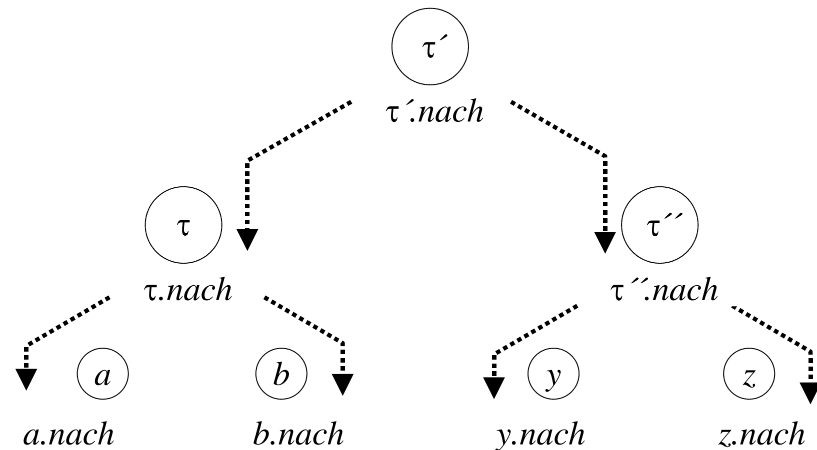
Fall B:

- Doppelter Baumdurchlauf:
  - zuerst von unten nach oben Schritte 1,2 durchführen (Mengen von *a priori* Typen berechnen)
  - dann von oben nach unten Schritte 3-5 (*a posteriori* Typ und Operation bestimmen, beides muß eindeutig sein)
  - unvermeidbar $LAG(2)$

5.2.2 Bestimmen der *a priori* Typen



5.2.2 *a posteriori* Typen



5.2.2 Zusätzlich

1. Typanpassung, z.B.

`IntType → RealType`

2. Implizites Dereferenzieren

`ReferenceType (IntType) → IntType`

3. Implizites Deprozedurieren von Funktionen ohne Argumente

`ProcedureType (Null, RealType) → RealType`

4. Typabgleich unter Berücksichtigung von 1 - 3,
   Anzahl dieser Operationen minimal.

`+(ReferenceType (IntType), ProcedureType(Null, RealType)) → + (RealType, RealType)`

Wie ist das Attribut ´Typ´ implementiert?

5.2.2 Schnittstelle für alle Typen

ADT Type

`getBaseType: Type → Type`
Gibt den Grundtyp eines Typen zurück

`getContentType: Type → Type`
Gibt den enthaltenen Typ zurück

`equivalent: Type × Type → boolean`
Wahr gdw. beide Typen strukturell gleich sind

`coercible: Type × Type → boolean`
Wahr gdw. der erste Typ in den zweiten umgewandelt werden kann

`balance: Type × Type → Type`
Typ in den beide durch implizite Dereferenzierung und Deprozedurierung umgewandelt werden können

5.2.2 Typabgleich und Typanpassung

Problem: bestimme richtigen (!) Zieltyp, an den Typen t, t´ angepaßt werden können.

Schemaverfahren: Unterscheide

- Grundtypen: einfache Typen, Reihung, Verbund, Prozedur mit Parametern, usw.
  - Grundtypen anpaßbar, wenn entweder gleich oder `int → real` Konversion usw.
- abgeleitete Typen: `ref t, proc t, ref ref t, ref proc t, proc proc t` usw.

Verfahren:

1. Bestimme Grundtyp,

2a. wenn Grundtyp(t) = Grundtyp(t´), ermittle Zieltyp durch Streichen einer minimalen Anzahl von Vorsilben `ref, proc`

2b. wenn Grundtyp(t) ≠ Grundtyp(t´) dann passe t an t´an oder umgekehrt, oder an Typ t´´, an den beide anpaßbar, oder Fehler

5.2.2 Beispiel

```
x: proc int;
y: ref proc int;
z: proc proc int;
x = expr ? y : z;
```

balance(y,z) ?

Grundtyp(y) = Grundtyp(z) = int

Fall 2a: streiche ref von y und proc von z

Ergebnis: proc int

5.2.2 Operatoridentifikation

Operatoridentifikation ist signaturabhängige Namensanalyse für Funktionen

Funktionen müssen für die Basistypen der Operanden, also *op.vor* dereferenziert und deprozeduriert definiert werden.

a posteriori Typ der Operanden und damit die notwendigen Typanpassungen werden durch die Operatoridentifikation mit festgelegt

sonst keine Neuigkeiten

Hinweis: Operatoridentifikation kann auch als Verfahren zur Codeerzeugung aufgefaßt werden: Jede Operation definiert ein oder mehrere Makros in Maschinensprache, die, wenn die Operanden eingesetzt sind, die Operation korrekt implementieren. Keine Eindeutigkeit bei der Auswahl gefordert, stattdessen Verwendung von Kostenfunktionen

Kapitel 5: Semantische Analyse

0. Einbettung
1. Namensanalyse
    1.1 Implementierung
2. Typanalyse und Operatoridentifikation
    2.1 Typanalyse
    2.2 Zusammenhänge
    2.3 Implementierung / Beispiel

5.2.3 Beispiel

Die hier beschriebene Sprache ist in mehrerer Hinsicht kurios:

- Implizites Dereferenzieren und Deprozedurieren
- Prozeduren können (ausführbare) Prozeduren zurückgeben. Achtung das ist mehr als der z.B. in C übliche Prozdurzeiger.
- Zwei Arten von Gleichheit:
  Die Identität „==“ nur auf ref anwendbar und
  strukturelle Gleichheit „=“ auf alles andere.
- ```
  x := 3;
  proc pi : ref int; pi=&x; end;
  t1 := a=pi;   // wahr wenn z.B. real a := 3.0
  t2 := a==pi;  // falsch selbst wenn int a := 3
  t3 := x==pi   // wahr (da selbe Referenz)
  proc bf : (proc : ref int); bf = pi; end;
  // nun könnte man bei den Vergleichen auch bf verwenden
  ```

  ```

# 5.2.3 Typ

```
abstract class Type {
  public final static Type VOID_TYPE = new Type() {};
  public final static Type BAD_TYPE = new Type() {};

  public Type getContentType() { return this; }

  public boolean isStrippable() { return false; }

  public Type getBaseType() {
    Type t = this;
    while (t.isStrippable()) t = t.getContentType();
    return t;
  }
  public Object clone() { return this; }
}
```

5.2.3 Basistypen

```
abstract class PrimitiveType extends Type {
  public final static PrimitiveType INT_TYPE =
    new PrimitiveType() {};

  public final static PrimitiveType REAL_TYPE =
    new PrimitiveType() {};

  public final static PrimitiveType BOOLEAN_TYPE =
    New PrimitiveType() {};
}
```

5.2.3 „Eingewickelte" Typen

```
abstract class WrappedType extends Type {
   private Type contentType;

   protected WrappedType(Type contentType)
     { this.contentType = contentType; }

   public Type getContentType()
     { return contentType; }

   public void setBaseType(Type new_base)
   {
      WrappedType t = this;
      while (t.isStrippable() && t.contentType.isStrippable())
         t = (WrappedType)t.contentType;
      t.contentType = new_base;
   }
}
```

5.2.3 Referenzen

```
Class ReferenceType extends WrappedType {
  public ReferenceType(Type baseType) {
    super(baseType);
  }

  public final static ReferenceType NIL_TYPE =
    new ReferenceType(Type.VOID_TYPE);

  public boolean isStrippable() { return true; }

  public Object clone()
  {
    return new ReferenceType((Type)getContentType().clone());
  }
}
```

5.2.3 Prozeduren

```
class ProcedureType extends WrappedType {
  private Type[] parameterTypes;
  public ProcedureType(Type contentType,
                       Type[] parameterTypes) {
    super(contentType);
    this.parameterTypes = parameterTypes;
  }
  public int getParameterCount()
    { return parameterTypes.length; }
  public Type getParameterType(int i)
    { return parameterTypes[i]; }
  public boolean isStrippable()
    { return getParameterCount()==0; }
  public Object clone()
  {
    return new ProcedureType((Type)getContentType().clone(),
                             parameterTypes);
  }
}
```

5.2.3 Reihungen

```
class ArrayType extends WrappedType {
  private int arity;

  public ArrayType(Type contentType, int arity) {
    super(contentType);
    this.arity = arity;
  }

  public int getArity() { return arity; }

  public Object clone()
  {
    return new ArrayType((Type)getContentType().clone(), arity);
  }
}
```

5.2.3 Verbunde

```
class RecordType extends Type {
  private Type[] compoundTypes;

  public RecordType(Type[] compoundTypes)
    { this.compoundTypes = compoundTypes; }

  public int getCompoundCount()
    { return compoundTypes.length; }

  public Type getCompoundType(int i)
    { return compoundTypes[i]; }
}
```

5.2.3 Typisierungsregeln

```
public class TypeRules {

  private TypeRules() {}

  public final static TypeRules RULES = new TypeRules();

  public boolean equivalent(Type a, Type b) {
    if (a == Type.BAD_TYPE || b == Type.BAD_TYPE) return false;
    if (a == b) return true;
    if (a.getClass() != b.getClass()) return false;
    if (a.isStrippable()) {
      return equivalent(a.getContentType(),b.getContentType());
    }
// equivalent wird fortgesetzt ...
```

5.2.3 Typisierungsregeln

```
  if (a instanceof ArrayType)
    return ((ArrayType)a).getArity()==((ArrayType)b).getArity();
  if (a instanceof ProcedureType) {
    ProcedureType pa = (ProcedureType)a;
    ProcedureType pb = (ProcedureType)b;
    int arity = pa.getParameterCount();
    if (pb.getParameterCount() != arity) return false;
    for (int i = 0; i < arity; i++) {
      if (!equivalent(pa.getParameterType(i),
      pb.getParameterType(i))) return false;
    }
    return equivalent(pa.getContentType(), pb.getContentType());
  }
  // optional (see procedure type above):
  // structural equivalence of RecordTypes:
  // compare all compound types
  return false;
} // TypeRules wird fortgesetzt ...
```

5.2.3 Typisierungsregeln

```
public boolean coercible(Type a, Type b) {
  if (a == Type.BAD_TYPE || b == Type.BAD_TYPE) return false;

  if (equivalent(a, b)) return true;

  if (a == PrimitiveType.INT_TYPE)
    return (b == PrimitiveType.REAL_TYPE);

  if (a == ReferenceType.NIL_TYPE)
    return (b instanceof ReferenceType);

  return false;
}

// TypeRules wird fortgesetzt ...
```

5.2.3 Typisierungsregeln

```
  private Type clone(Type a, Type new_base)
  {
    if (! a.isStrippable())
      return new_base;

    WrappedType res = (WrappedType)a.clone();
    res.setBaseType(new_base);
    return res;
  }

// TypeRules wird fortgesetzt ...
```

5.2.3 Typisierungsregeln

```
public Type balance(Type a, Type b)
{
  Type a_rest, b_rest;

  if(! equivalent(a.getBaseType(), b.getBaseType()) )
    if (coercible(a.getBaseType(), b.getBaseType()))
      a = clone(a, b.getBaseType());
    else if (coercible(b.getBaseType(), a.getBaseType()))
      b = clone(b, a.getBaseType());
    else
      return Type.BAD_TYPE;

// balance wird fortgesetzt ...
```

5.2.3 Typisierungsregeln

```
      for (a_rest = a;; a_rest = a_rest.getContentType())
      {
        for (b_rest = b;; b_rest = b_rest.getContentType())
        {
          // if one of them can be converted into the other,
          // it's the "bigger" one
          if (coercible(a_rest, b_rest)) return b_rest;
          if (coercible(b_rest, a_rest)) return a_rest;

          if (!b_rest.isStrippable())
            break;
        }
        if (!a_rest.isStrippable())
          break;
      }
      return Type.BAD_TYPE;
    }
  }
```

5.2.3 Typisierung von Ausdrücken

```
rule zuweisung::= name´:=´ausdruck.
attribution
       zuweisung.vor ← name.nach;
       name.nach ← name.vor.deproc();
       ausdruck.nach ←
              (!name.nach instanceof ReferenceType) ?
                     Type.VOID_TYPE :
                     name.nach.target;
condition
       coercible(zuweisung.vor, zuweisung.nach)&&
       name.nach == instanceof ReferenceType;

rule ausdruck::= vergleich.
attribution
       ausdruck.vor ← vergleich.vor;
       vergleich.nach ←ausdruck.nach;
condition
       coercible(ausdruck.vor, ausdruck.nach)
```

5.2.3 Typisierung von Ausdrücken II

```
rule vergleich ::= ausdruck verglop ausdruck.
attribution
   vergleich.vor ← PrimitiveType.BOOLEAN_TYPE;
   ausdruck[1].nach ← verglop.nach;
   ausdruck[2].nach ← verglop.nach;
   verglop.vor ←balance(ausdruck[1].vor, ausdruck[2].vor);
condition
  coercible(vergleich.vor, vergleich.nach);

rule verglop ::= ´=´.
attribution
  verglop.nach ← verglop.vor.deref();
condition
  ! verglop.nach instanceof Type.VOID_TYPE;
```

5.2.3 Typisierung von Ausdrücken III

```
rule verglop ::= ´==´.
attribution
  verglop.nach ← verglop.vor.deproc();
condition
  verglop.nach instanceof ReferenceType;

rule ausdruck::= name.
attribution
  ausdruck.vor ← name.vor;
  name.nach ←ausdruck.nach;
condition
  coercible(ausdruck.vor, ausdruck.nach);
```

5.2.3 Typisierung von Ausdrücken IV

```
rule name ::= bezeichner.
attribution
   name.vor ← Type.ANY_TYPE;
condition
   coercible(name.vor, name.nach);

rule name ::= name´.´bezeichner.
Attribution
   name[2].umg ← name[1].umg;
   name[2].nach ← name[1].vor;
   name[1].vor ←
      name[1].vor instanceof RecordType ?
        new Type(name[2].umg.definitionInRange(name[1].vor,bezeichner)):
        new BadType();
condition
   coercible(name[1].vor, name[1].nach);
```