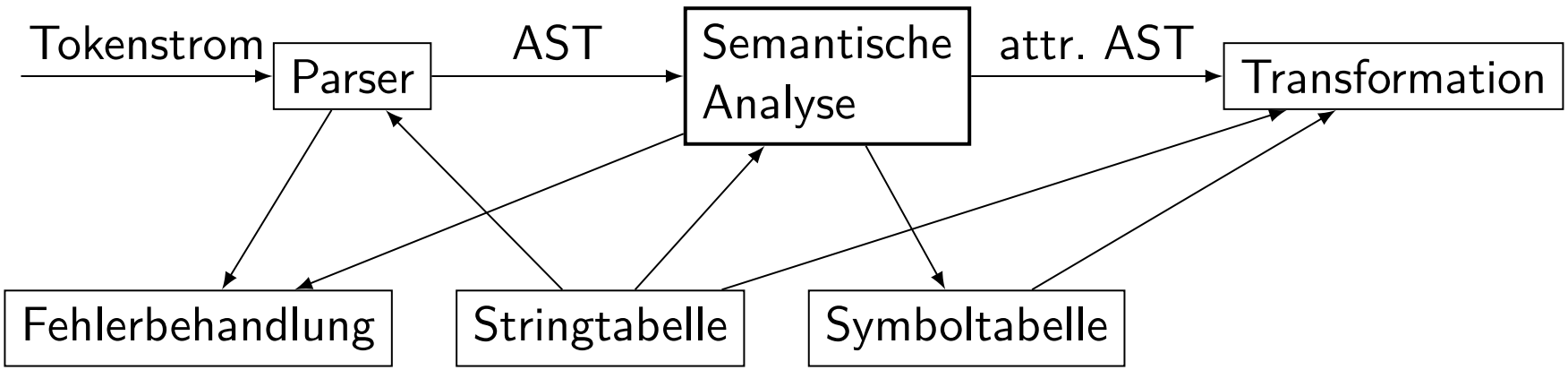
Kapitel 5

Semantische Analyse

Kapitel 5: Semantische Analyse

Eingliederung in den Übersetzer



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?

Kapitel 5: Semantische Analyse

Namensanalyse

Unterscheide:

- Bezeichnerdefinition (*defining occurrence*): Vereinbarung, Parameterspezifikation, Markendefinition, vordefiniert, ...
  - vordefiniert: definiert in einem das Gesamtprogramm umfassenden 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, C)

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

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).

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)

Eigenschaften von typ: siehe Beispiel-AGs in Kap. 4

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

Blockschachtelung: nur vorangehende Vereinbarungen

1. **rule** block → deklarationen 'begin' anweisungen 'end' .
   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 $\mathrm{LAG}(1)$ und OAG

Blockschachtelung: nur vorangehende Vereinbarungen

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.

Blockschachtelung: Verwendung vor Vereinbarung

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

Zweifacher Durchlauf:

1. Definitionen zusammenführen (synthetisieren) in *umg_part*. Kombination aller Definitionen an der Wurzel ergibt *umg*
2. Definitionen aus ererbtem *umg* verwenden

Schema ist $\mathrm{LAG}(2)$ und OAG

Blockschachtelung: Verwendung vor Vereinbarung

rule block → deklarationen ';' anweisungen .
attribution
deklarationen.umg := **append**(block.umg, deklarationen.umg_part);
anweisungen.umg := deklarationen.umg;

rule deklarationen → deklarationen deklaration .
attribution
deklarationen[1].umg_part := **append**(deklarationen[2].umg_part, deklaration.umg_part);
deklarationen[2].umg := deklarationen[1].umg;
deklaration.umg := deklarationen[1].umg;

rule deklaration → bezeichner ':' typ ':=' expression ';' .
attribution
deklaration.dekl := **new** Deklaration();
deklaration.umg_part := **new** Umg((bezeichner.symbol, deklaration.dekl));
deklaration.dekl.typ := deklaration.umg.search(typ.symbol);
deklaration.dekl.expr := expression.expr;
expression.umg := deklaration.umg;

rule expression → bezeichner ';' .
attribution
expression.expr := **new** Readvar();
expression.expr.vardekl := expression.umg.search(bezeichner.symbol);

Einschub: Zyklische Abhängigkeiten - Fehlerhafter Versuch

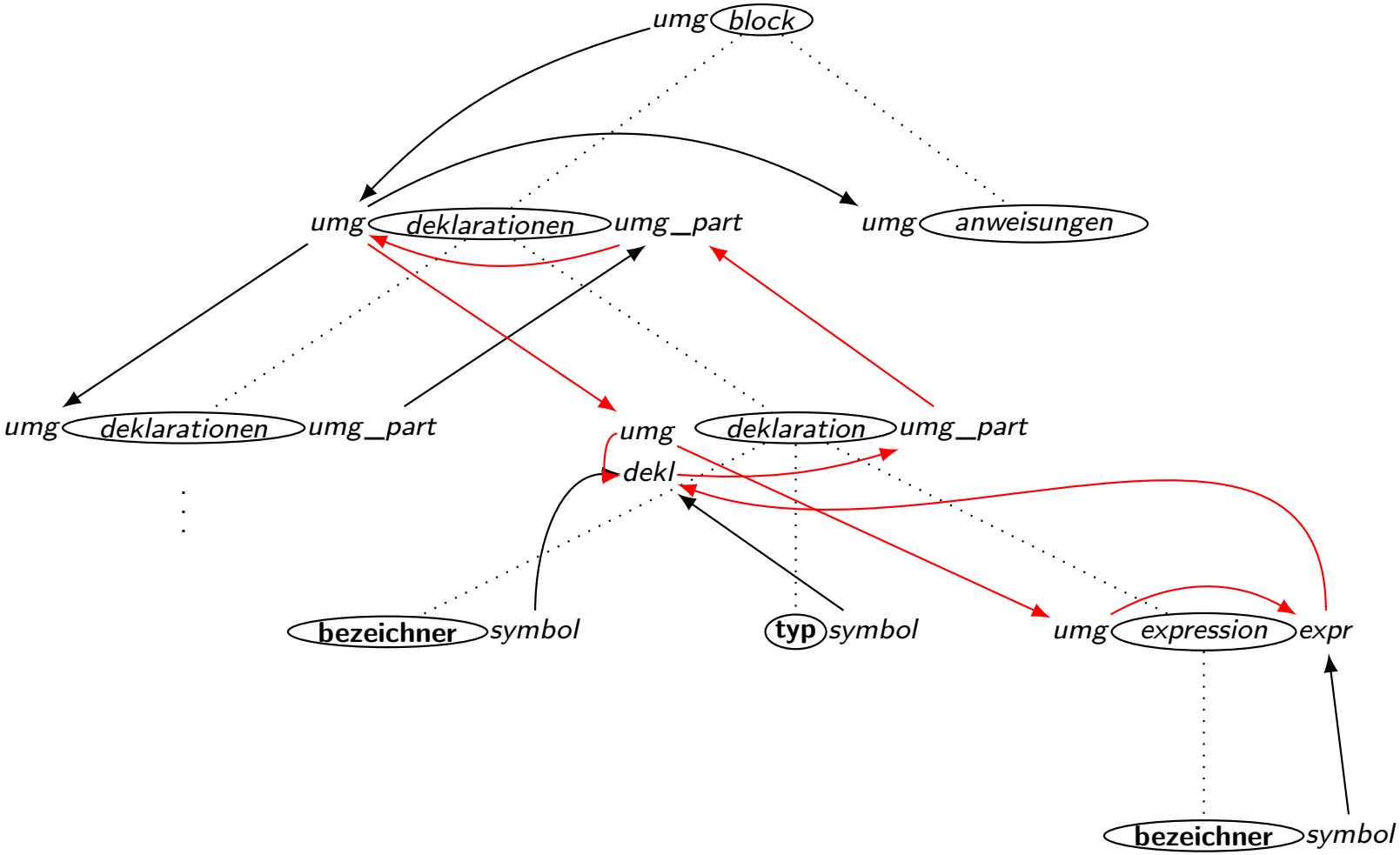
rule block → deklarationen ';' anweisungen .
attribution
deklarationen.umg := **append**(block.umg, deklarationen.umg_part);
anweisungen.umg := deklarationen.umg;

rule deklarationen → deklarationen deklaration .
attribution
deklarationen[1].umg_part := **append**(deklarationen[2].umg_part, deklaration.umg_part);
deklarationen[2].umg := deklarationen[1].umg;
deklaration.umg := deklarationen[1].umg;

rule deklaration → bezeichner ':' typ ':=' expression ';' .
attribution
deklaration.dekl := **new** Deklaration(
    bezeichner.symbol, deklaration.umg.search(typ.symbol),
    expression.expr);
deklaration.umg_part := **new** Umg((bezeichner.symbol, deklaration.dekl));
expression.umg := deklaration.umg;

rule expression → bezeichner ';' .
attribution
expression.expr := **new** Readvar();
expression.expr.vardekl := expression.umg.search(bezeichner.symbol);

Zyklische Abhängigkeit - Beispiel an konkretem AST



Wiederholung: Korrekte Lösung

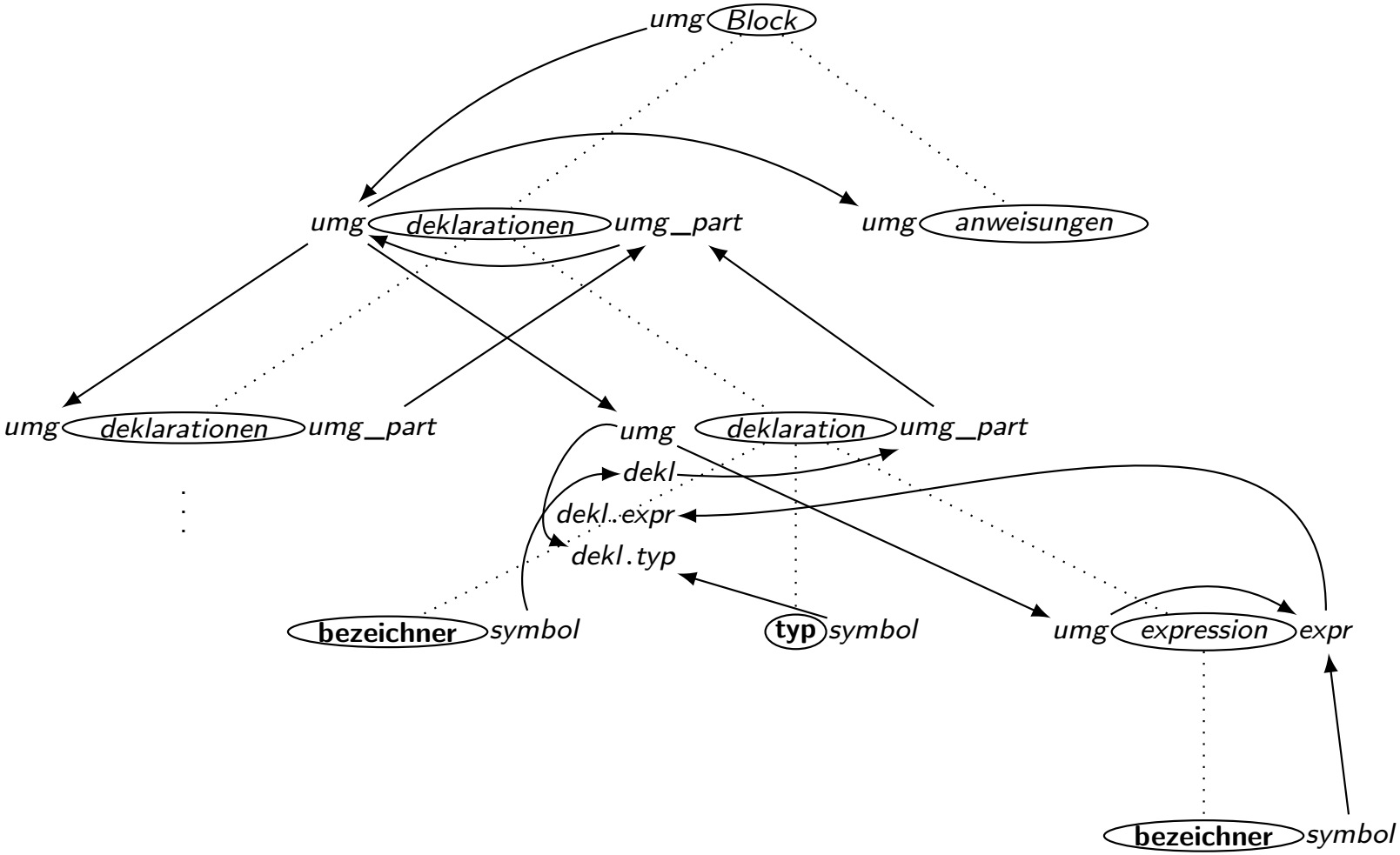
rule block → deklarationen ';' anweisungen .
attribution
deklarationen.umg := **append**(block.umg, deklarationen.umg_part);
anweisungen.umg := deklarationen.umg;

rule deklarationen → deklarationen deklaration .
attribution
deklarationen[1].umg_part := **append**(deklarationen[2].umg_part, deklaration.umg_part);
deklarationen[2].umg := deklarationen[1].umg;
deklaration.umg := deklarationen[1].umg;

rule deklaration → bezeichner ':' typ ':=' expression ';' .
attribution
deklaration.dekl := **new** Deklaration();
deklaration.umg_part := **new** Umg((bezeichner.symbol, deklaration.dekl));
deklaration.dekl.typ := deklaration.umg.search(typ.symbol);
deklaration.dekl.expr := expression.expr;
expression.umg := deklaration.umg;

rule expression → bezeichner ';' .
attribution
expression.expr := **new** Readvar();
expression.expr.vardekl := expression.umg.search(bezeichner.symbol);

Reparierte AG



Pascal, Modula-2

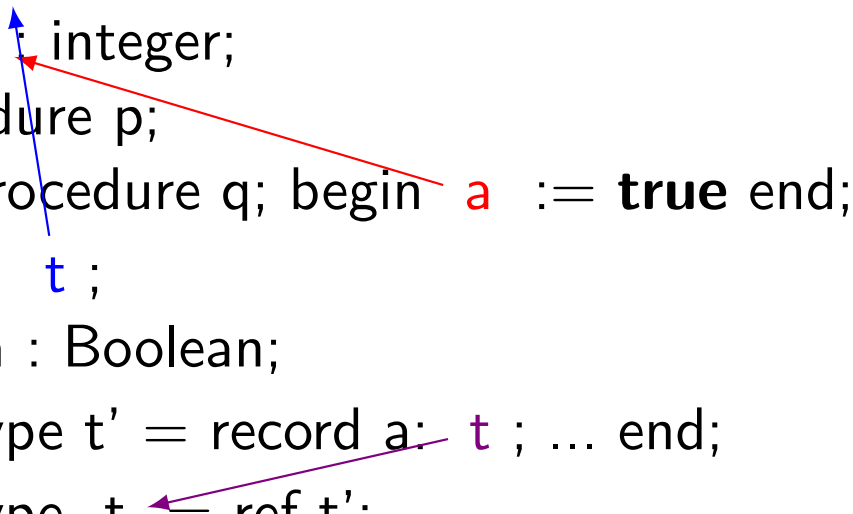
Definitionen von a, t, t:

```
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 *)
```

Pascal, Modula-2

Definitionen von a, t, t: Häufigste Lösung:

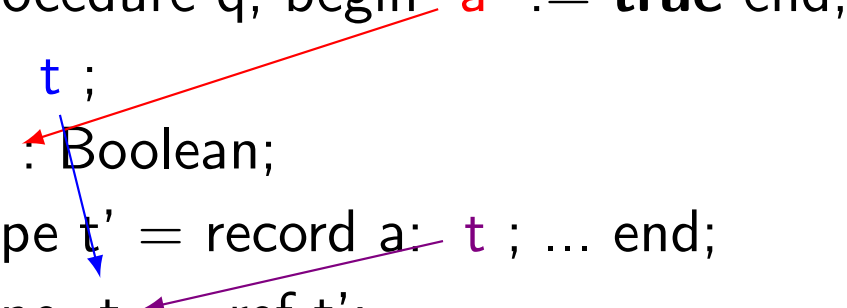
```
type  t  = ...
var  a : integer;
procedure p;
      procedure q; begin  a  := true end;
      b:  t ;
       a : Boolean;
      type t' = record a:  t ; ... end;
      type  t  = ref t';
      ...
begin (* p *)
...
end; (* p *)
```



Pascal, Modula-2

Definitionen von a, t, t: Korrekte Lösung:

```
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 *)
```



Verbundfelder in Pascal

Zugriff auf Feld a: x.a oder **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 Modulen, Klassen, Objekten, ... analog

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

- Zusatzprobleme 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 Modulen: 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

Teilweise Qualifikation

In COBOL, PL/1: Wenn Verbund x Unterverbunde $p, \ldots$ 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

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;
```

Ergebnis der Namensanalyse

- ein Eintrag für jede Definition in der Symboltabelle
- Verweis auf diesen Eintrag für jede Anwendung
- Symboltabelle: die zentrale „Datenbank“ des Übersetzers, unstrukturierte Menge von Definitionseinträgen, auch Bestandteil der Metadaten in .NET-Päckchen

Implementierung der Namensanalyse

Hauptaufgabe: Effiziente Implementierung von umg:
Namenstabelle

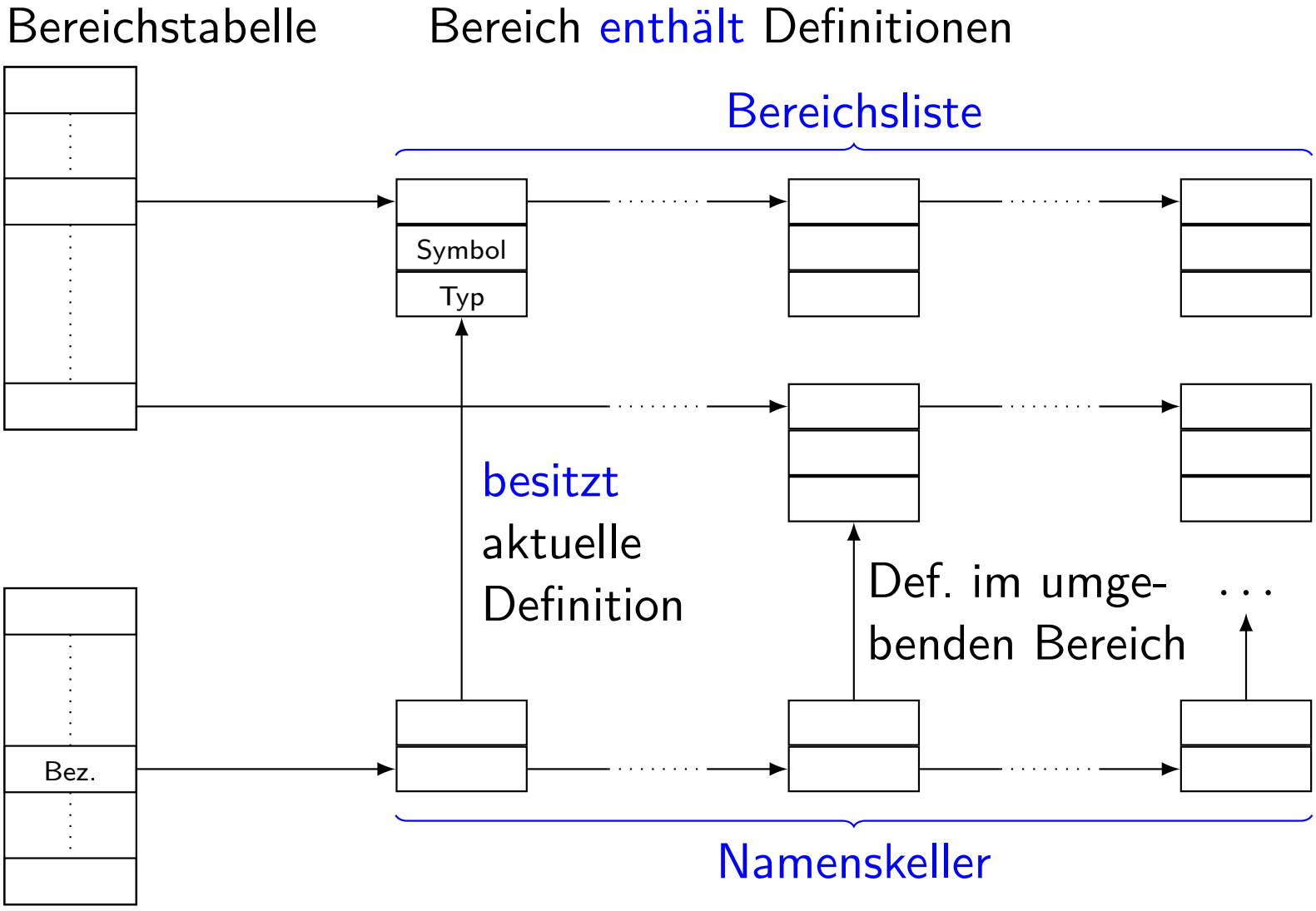
- Suchen in der Umgebung vermeiden, Ziel O(1)
- Namenstabelle als zentrale Datenstruktur außerhalb des Strukturbaums speichern
  - Aufbau unabhängig von Besuchssequenzen
  - Bei Eintritt/Verlassen eines Namensraums ändert sich der gültige Teil der Tabelle

Symboltabelle

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
- Nach Analyseende ist Namenskeller wieder leer, er wird nur während der Analyse verwendet.

Symboltabelle



ADT Symboltabelle

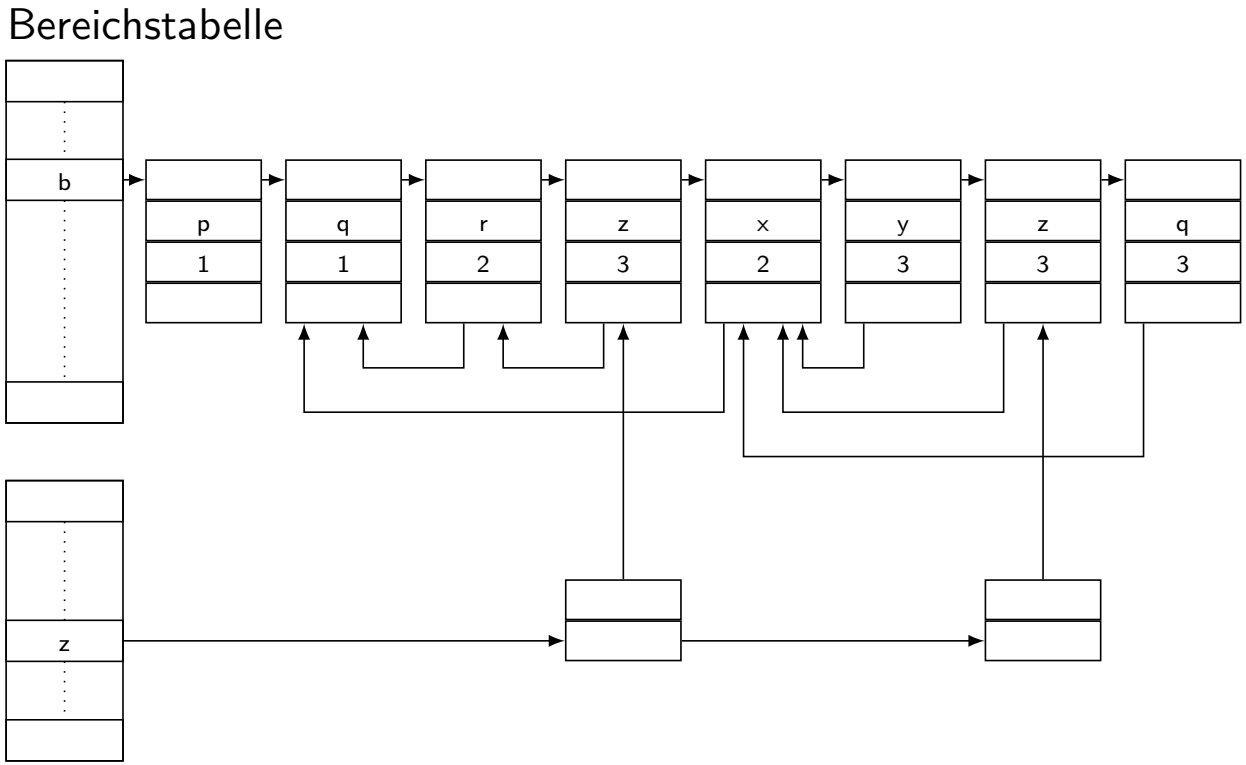
```
abstract class SymbolTable {
    private Range currentRange;

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

Namenstabelle II

Erkenntnis: Namenskeller aus der Stringtabelle 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

Namenstabelle: teilweise Qualifikation




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

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;
}
```

Vordefinierte Namen (1/2)

1. Gewöhnliche Definitionen für alle Programme gültig: Vorbesetzung Namens- und Symboltabelle
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

Vordefinierte Namen (2/2)

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 Prozedurwerte u.ä. zulässig! Das steht oft nicht im Sprachbericht!

Zusammenfassung Namensanalyse

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

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

Typanalyse (1/2)

Typ: Kennzeichnung von Objekten bezüglich zulässiger Wertemenge und zulässigen Operationen

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

Aufgabe der Typanalyse:

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

Typanalyse (2/2)

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!

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.

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

Typattribute

bereits bekannt:

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

Typ von Namen, Objekten in Symboltabelle 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

Zusammenspiel Typanalyse - Operatoridentifikation

1. a priori Typ der Operanden von $op1\ t\ op2$ übernehmen
2. mögliche Definitionen von t 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 $\longrightarrow$ 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

Eindeutigkeit von Operator- und Funktionsdefinitionen

nachgeprüft bei Eintrag in die Symboltabelle (Algol 68)

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

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

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

Typanalyse im Ausdrucksbaum

Unterscheide

1. Operatoridentifikation hängt nur von den Operandentypen ab (alle anderen Sprachen)
2. Operatoridentifikation hängt auch vom verlangten Ergebnistyp ab (linke Seite einer Zuweisung, Ada)

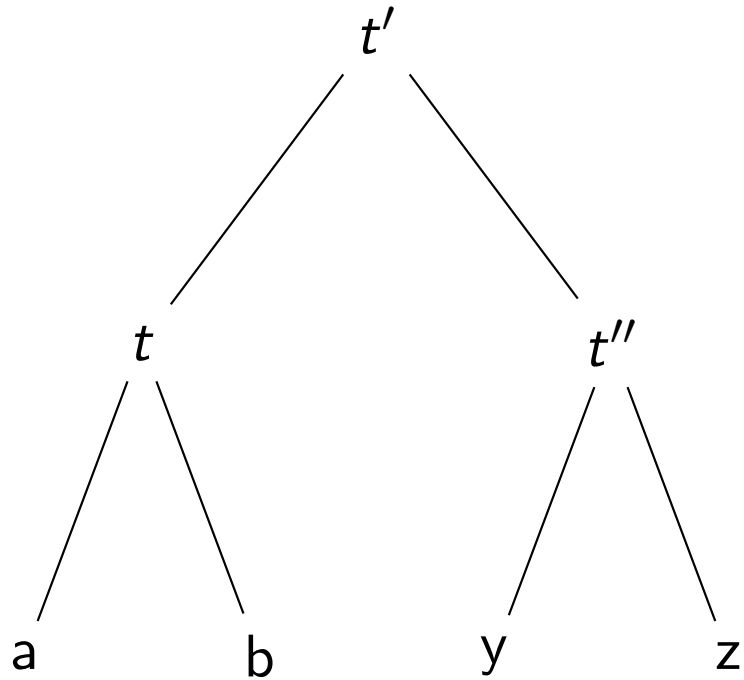
Fall 1:

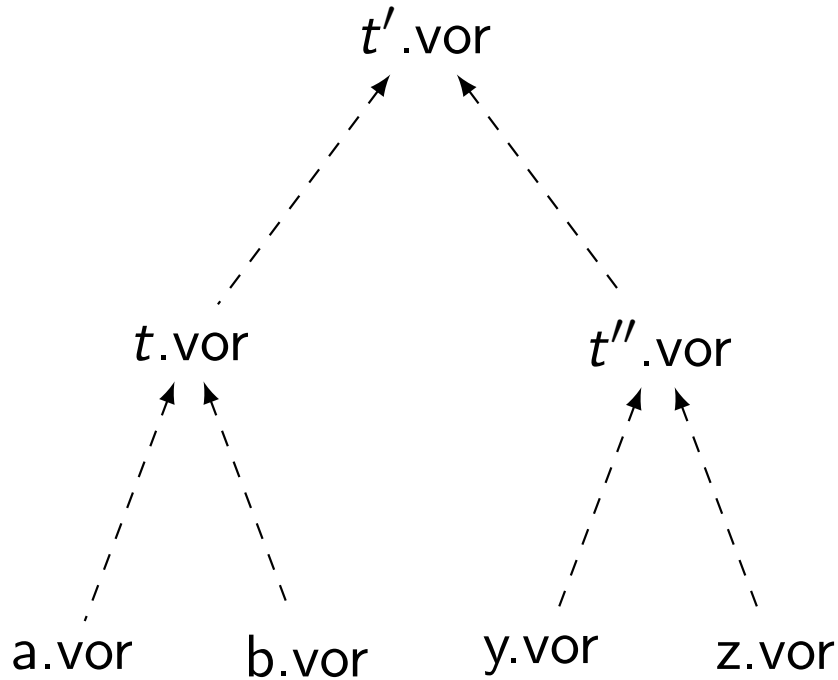
- Schritte 1-5 des Algorithmus von unten nach oben im Baum durchführen (bei Rückkehr aus Tiefensuche)
  - formal LAG(2), da ererbtes Attribut zu spät berechnet

Fall 2:

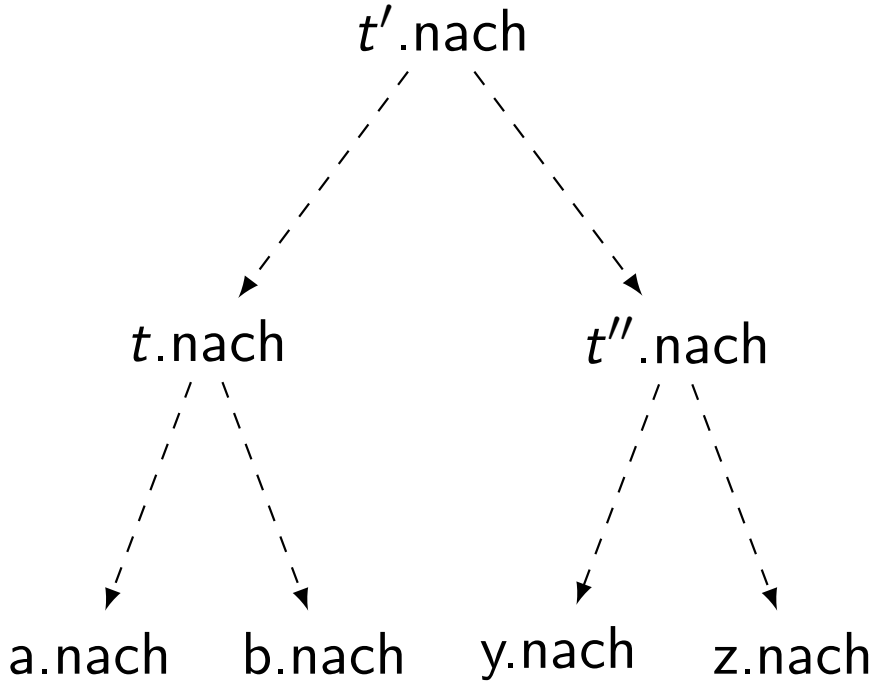
- Doppelter Baumdurchlauf:
  - zuerst von unten nach oben Schritte 1,2 durchführen (Mengen von a priori Typen berechnen)
  - dann von oben nach unten Schritte 3-5 (a posteriori Typ und Operation bestimmen, beides muss eindeutig sein)
  - unvermeidbar LAG(2)

Bestimmen der a priori Typen





Bestimmen der a posteriori Typen



Zusätzlich

1. Typanpassung, z.B. IntType $\rightarrow$ RealType
2. Implizites Dereferenzieren
   ReferenceType (IntType) $\rightarrow$ IntType
3. Implizites Deprozedurieren von Funktionen ohne Argumente
   ProcedureType (Null, RealType) $\rightarrow$ RealType
4. Typabgleich unter
   Berücksichtigung von 1 - 3, Anzahl dieser Operationen minimal.
   +(ReferenceType (IntType), ProcedureType(Null, RealType))
   $\rightarrow$ + (RealType, RealType)

Wie ist das Attribut 'Typ' implementiert?

Hinweis: Das hier und im folgenden verwendete Typsystem entspricht keiner realen Sprache, sondern ist eine „plakative Vereinigung“ vieler Sprachen.

Schnittstelle für alle Typen

ADT Type

- getBaseType: Type $\longrightarrow$ Type
  - Gibt den Grundtyp eines Typen zurück
- getContentType: Type $\longrightarrow$ Type
  - Gibt den enthaltenen Typ zurück
- equivalent : Type $\times$ Type $\longrightarrow$ boolean
  - Wahr gdw. beide Typen strukturell gleich sind
- coercible : Type $\times$ Type $\longrightarrow$ boolean
  - Wahr gdw. der erste Typ in den zweiten umgewandelt werden kann
- balance: Type $\times$ Type $\longrightarrow$ Type
  - Typ in den beide durch implizite Dereferenzierung und Deprozedurierung umgewandelt werden können

Typabgleich und Typanpassung

Problem: bestimme richtigen (!) Zieltyp, an den Typen t, t' angepasst werden können.
Schemaverfahren: Unterscheide

- Grundtypen: einfache Typen, Reihung, Verbund, Prozedur m. Parametern, ...
  - Grundtypen anpaßbar, wenn gleich oder int $\rightarrow$ real Konversion usw.
- abgeleitete Typen: ref t, proc t, ref ref t, ref proc t, usw.

Verfahren:

1. Bestimme Grundtyp,
2. wenn Grundtyp(t) = Grundtyp(t'), ermittle Zieltyp durch Streichen einer minimalen Anzahl von Vorsilben ref, proc
3. wenn Grundtyp(t) $\neq$ Grundtyp(t') dann passe t an t' an oder umgekehrt, oder an Typ t'', an den beide anpaßbar, oder Fehler

Uniformer Zugriff

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

balance(y,z) ?

Grundtyp(y) = Grundtyp(z) = int
Fall 3: streiche ref von y und proc von z
Ergebnis: proc int

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

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

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; }
}
```

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() {};
}
```

„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;
    }
}
```

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());
  }
}
```

Prozeduren (1/2)

```
class ProcedureType extends WrappedType {
  private Type[] parameterTypes;

  public ProcedureType(Type contentType, Type[] parameterTypes) {
    super(contentType);
    this.parameterTypes = parameterTypes;
  }

  public int getParameterCount() {
      return parameterTypes.length;
  }

  ...
```

Prozeduren (2/2)

```
    ...

    public Type getParameterType(int i) {
        return parameterTypes[i];
    }

    public boolean isStrippable() {
        return getParameterCount()==0;
    }

    public Object clone() {
      return new ProcedureType((Type)getContentType().clone(),
                                parameterTypes);
    }
}
```

Arrays

```
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)
  }
}
```

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]; }
}
```

Typisierungsregeln (1/6)

```
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());
    }
    if (a instanceof ArrayType)
        return ((ArrayType)a).getArity()==((ArrayType)b).getArity();
// equivalent wird fortgesetzt ...
```

Typisierungsregeln (2/6)

```
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 ...
```

Typisierungsregeln (3/6)

```
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 ...
```

Typisierungsregeln (4/6)

```
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 ...
```

Typisierungsregeln (5/6)

```
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 ...
```

Typisierungsregeln (6/6)

```
    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;
} }
```

Typisierung von Ausdrücken (1/8)

rule zuweisung → 'name' := ausdruck .
attribution
    zuweisung.vor := name.nach;
    name.nach := name.vor.deproc();
    ausdruck.nach :=
        if !name.nach **instanceof** ReferenceType
        then Type.VOID_TYPE **else** name.nach.target;
condition
    coercible(zuweisung.vor, zuweisung.nach) &&
    name.nach == **instanceof** ReferenceType;

Typisierung von Ausdrücken (2/8)

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

Typisierung von Ausdrücken (3/8)

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);

Typisierung von Ausdrücken (4/8)

rule verglop → '=' .
attribution
    verglop.nach := verglop.vor.deref();
condition
    ! verglop.nach **instanceof** Type.VOID_TYPE;

Typisierung von Ausdrücken (5/8)

rule verglop → '==' .
attribution
    verglop.nach := verglop.vor.deproc();
condition
    verglop.nach **instanceof** ReferenceType;

Typisierung von Ausdrücken (6/8)

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

Typisierung von Ausdrücken (7/8)

rule name $\longrightarrow$ bezeichner .
attribution
    name.vor := Type.ANY_TYPE;
condition
    coercible(name.vor, name.nach);

Typisierung von Ausdrücken (8/8)

rule name → name '.' bezeichner .
attribution
    name[2].umg := name[1].umg;
    name[2].nach := name[1].vor;
    name[1].vor := **if** name[1].vor **instanceof** RecordType
        then new Type(
                name[2].umg.definitionInRange(name[1].vor,
                                              bezeichner))
        else new BadType();
condition
    coercible(name[1].vor, name[1].nach);