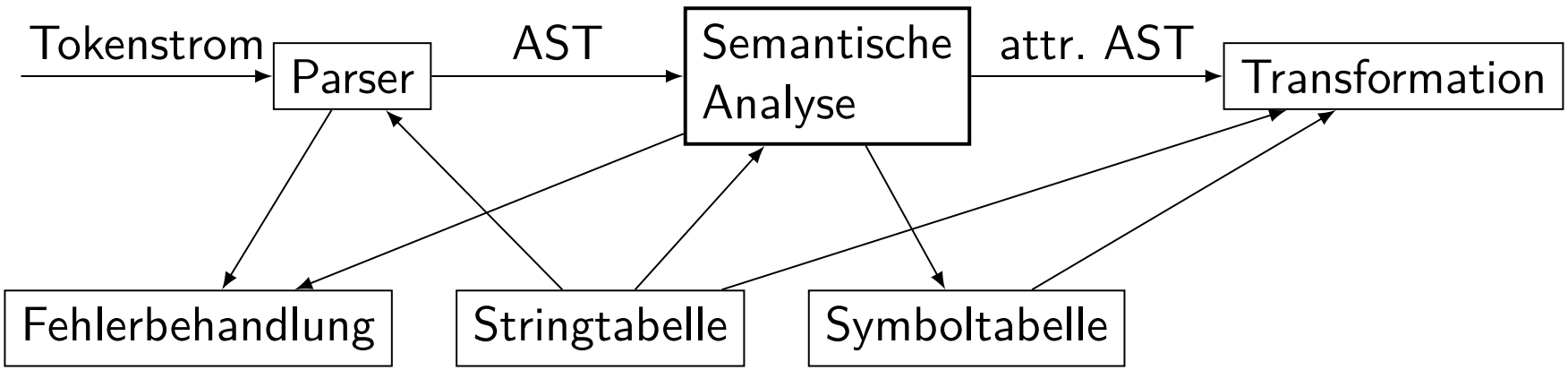
Kapitel 5

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 $\rightarrow$ Definition abhängig von Gültigkeitsbereichsregeln, syntaktischer Position: unterschiedliche Namensräume

Pascal, Modula-2

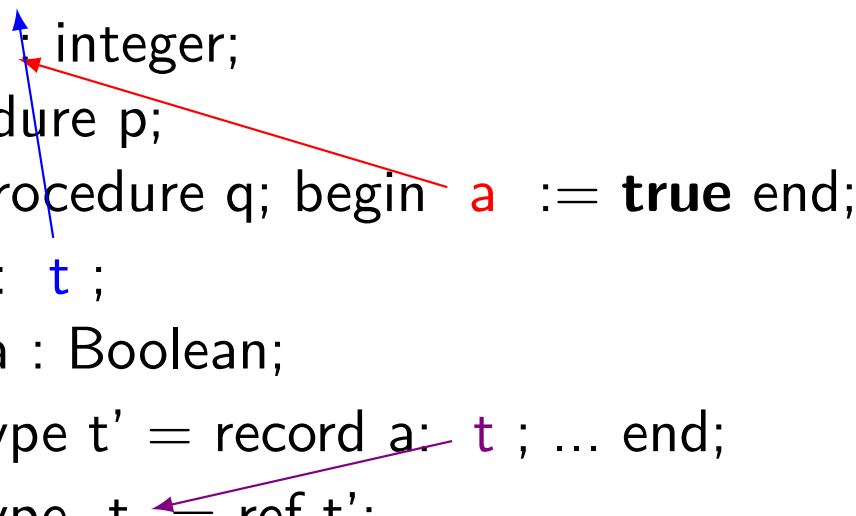
Definitionen von a, t, t:

```
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:

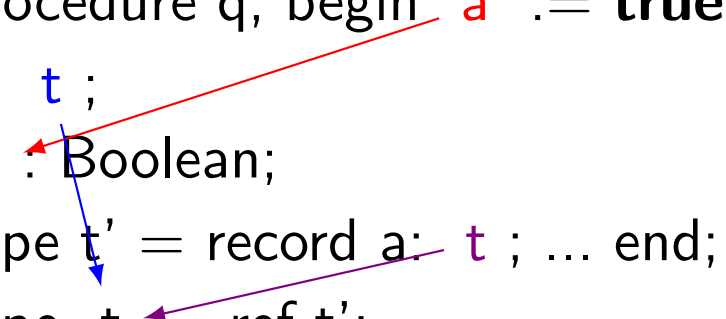
```
type  t  = ...
var  a : integer;
procedure p;
      procedure q; begin  a  := true end;
      b:  t ;
       a : Boolean;
      type t' = record a:  t ; ... end;
      type  t  = ref t';
      ...
begin (* p *)
...
end; (* p *)
```



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 *)
```



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

Zyklische Abhängigkeiten - Fehlerhafter Versuch

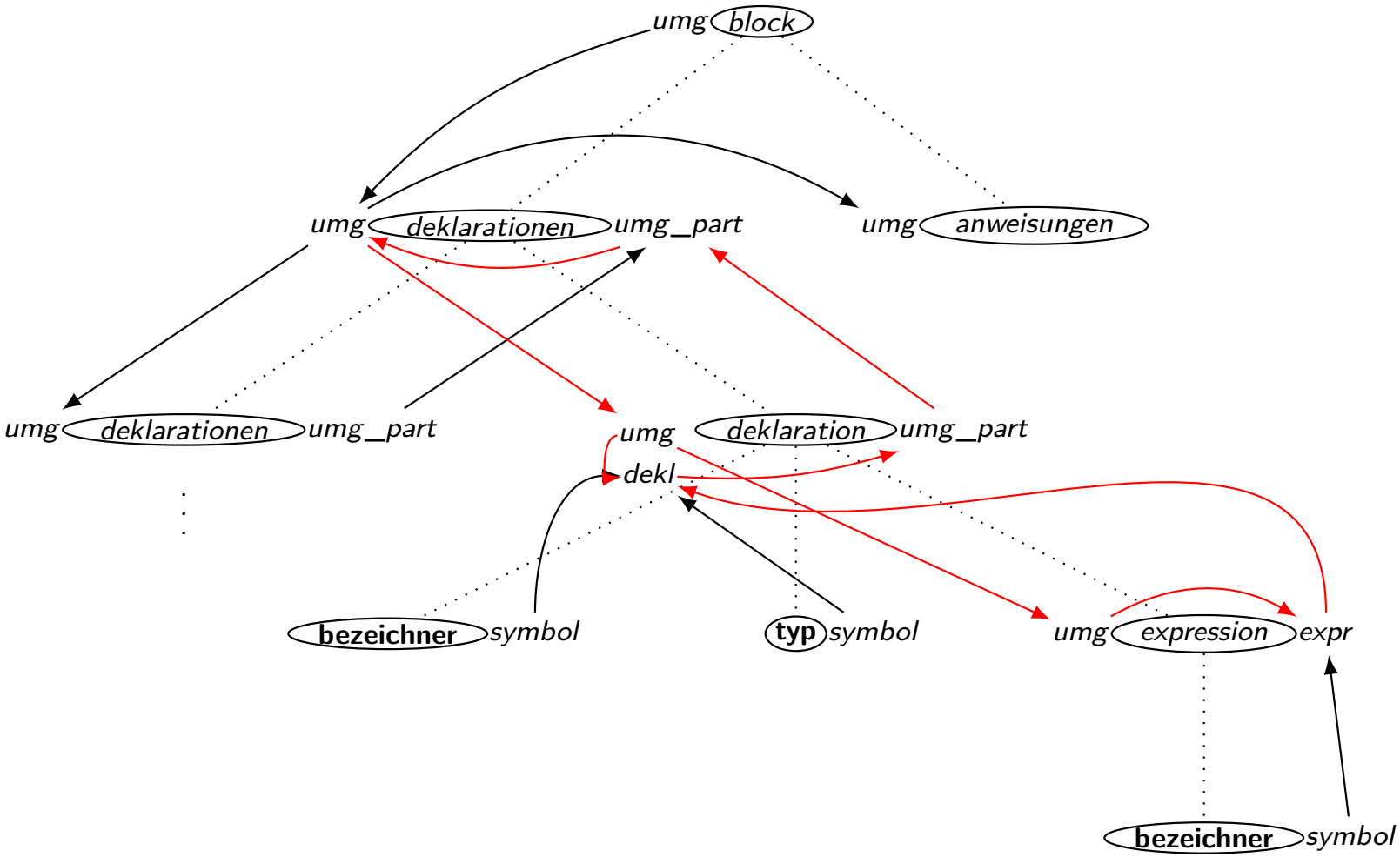
rule block → deklarationen ';' anweisungen .
attribution
deklarationen.umg := **append**(deklarationen.umg_part, block.umg);
anweisungen.umg := deklarationen.umg;

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



Blockschachtelung: Verwendung vor Vereinbarung

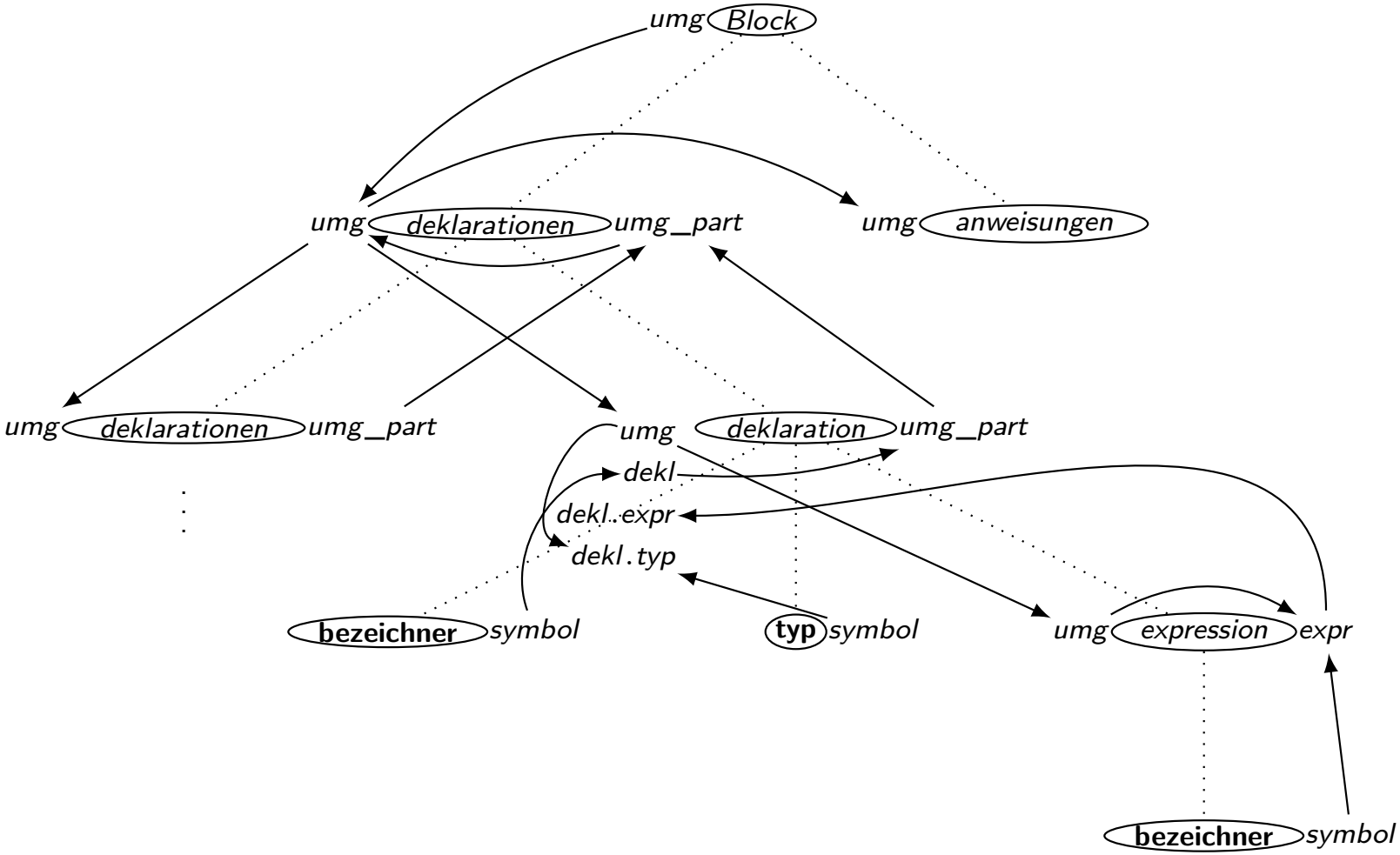
rule block → deklarationen ';' anweisungen .
attribution
deklarationen.umg := **append**(deklarationen.umg_part, block.umg);
anweisungen.umg := deklarationen.umg;

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



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

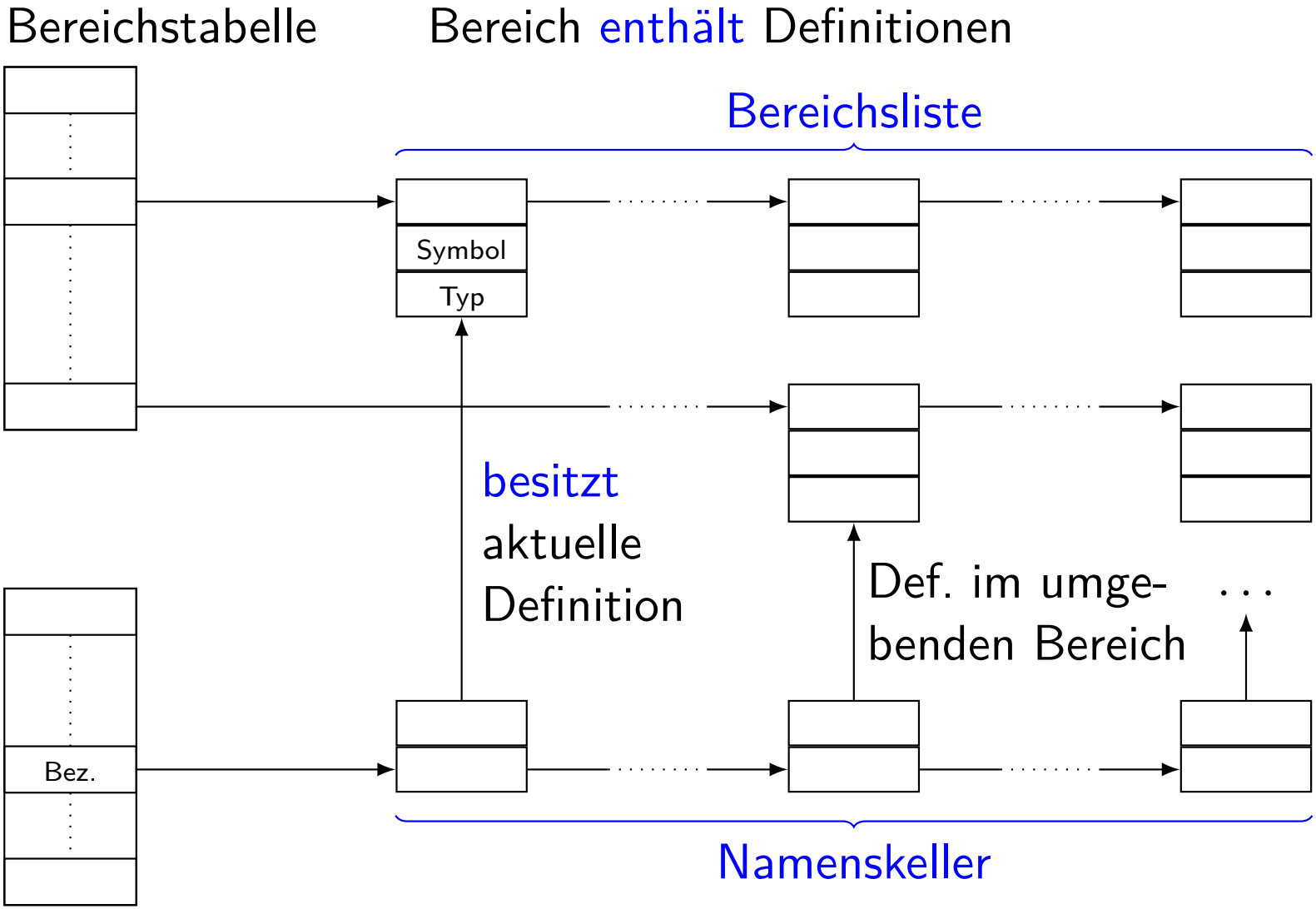
- Suchen in der Umgebung vermeiden, Ziel O(1)
- Namenstabelle als zentrale Datenstruktur außerhalb des Strukturbaums speichern
  - Aufbau unabhängig von Besuchssequenzen
  - Bei Eintritt/Verlassen eines Namensraums ändert sich der gültige Teil der Tabelle

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



Bez-Einträge in
Stringtabelle

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);
}
```

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

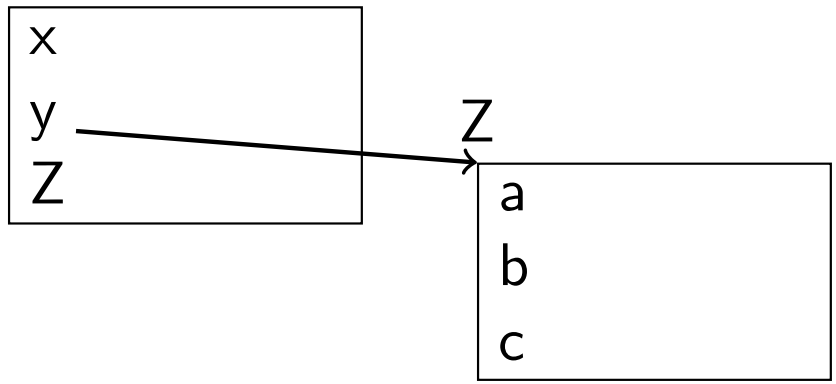
Bei Identifikation von x.a wird nur der Verbundtyp als Umgebung verwandt: keine Vererbung des äußeren Umgebungsattributs wie bei geschachtelten Blöcken

qualifizierter Zugriff x.a auf Attribute in Modulen, Klassen, Objekten, . . . analog

Benannte Gültigkeitsbereiche – Klassen

Grundidee: Am Klassennamen hängt komplette Symboltabelle
$\Rightarrow$ Schachtelung der Symboltabelle über Klassennamen

```
class Z {
   public int a;
   public int b;
   public int c;
}
int x;
Z y;
y.b = x;
```



Bei Analyse von y.f(a) wird unmittelbar hinter dem Punkt die Z-Tabelle verwendet, die über den Eintrag von y erreichbar ist. Für die Parameterliste a wird wieder der Symboltabellen-Stack (Blockschachtelung) verwendet.

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

1. Eingliederung in den Übersetzer
2. Namensanalyse
   - Allgemein
   - Beispiele aus der Praxis
   - Implementierung
3. Typanalyse und Operatoridentifikation
   - Typanalyse
   - Zusammenhänge

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!

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

Wiederholung: Typanpassung - Beispiel AG

rule Zuweisung → Name ':=' Ausdruck .
attribution
Name.umg := Zuweisung.umg;
Ausdruck.umg := Zuweisung.umg;
Name.nach := Name.vor;
Ausdruck.nach := **if** Name.vor = int **then** int **else** float **end**;

rule Ausdruck → Name addop Name .
attribution
Name[1].umg := Ausdruck.umg;
Name[2].umg := Ausdruck.umg;
Ausdruck.vor := **if** anpassbar(Name[1].vor, int) ∧ anpassbar(Name[2].vor, int)
  then int **else** float **end**;
addop.Typ := Ausdruck.vor;
Name[1].nach := Ausdruck.vor;
Name[2].nach := Ausdruck.vor;
condition anpassbar(Ausdruck.vor, Ausdruck.nach);

rule addop → '+' .
attribution addop.operation := **if** addop.Typ = int **then** int_add **else** float_add **end**;

rule Name → Bezeichner .
attribution Name.vor := definiert(Bezeichner.Symbol, Name.umg);
condition anpassbar(Name.vor, Name.nach);

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 $\rightarrow$ 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

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)

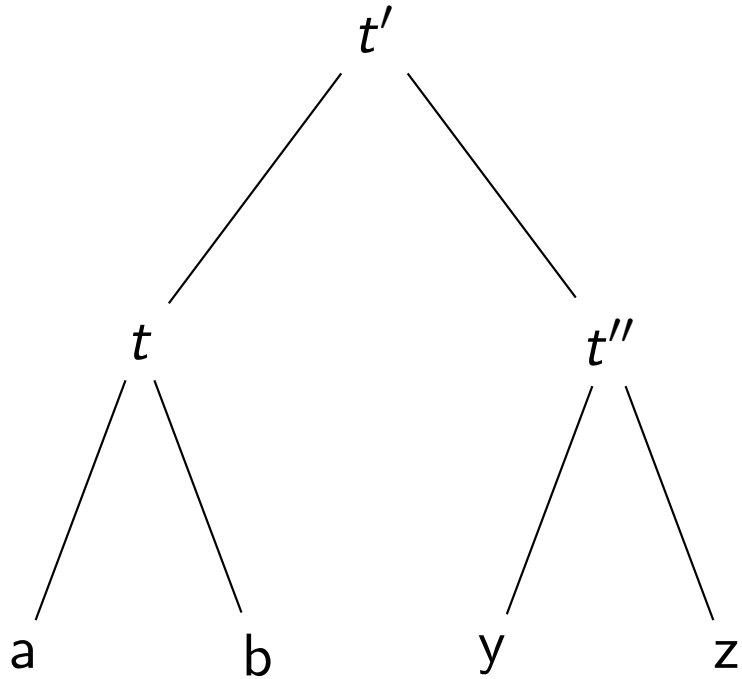
Fall 1:

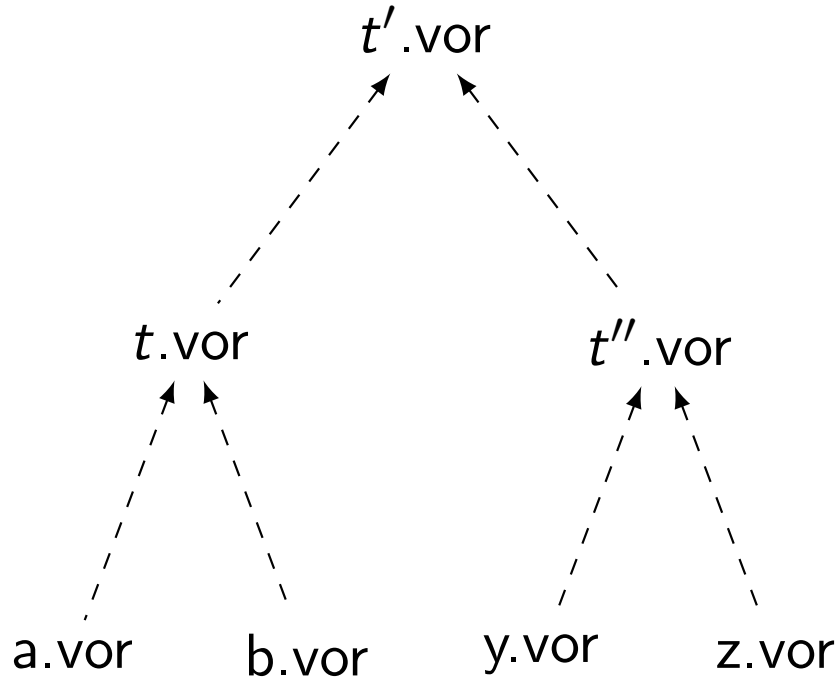
- Schritte 1-5 des Algorithmus von unten nach oben im Baum durchführen (bei Rückkehr aus Tiefensuche)
  - formal LAG(2), da ererbtes Attribut zu spät berechnet

Fall 2:

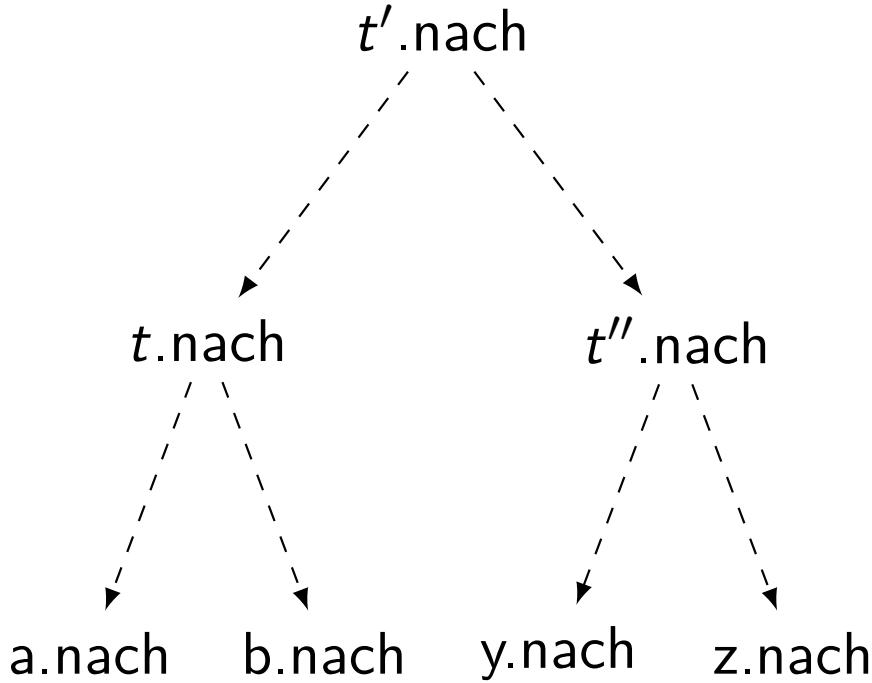
- Doppelter Baumdurchlauf:
  - zuerst von unten nach oben Schritte 1,2 durchführen (Mengen von a priori Typen berechnen)
  - dann von oben nach unten Schritte 3-5 (a posteriori Typ und Operation bestimmen, beides muss eindeutig sein)
  - unvermeidbar LAG(2)

Bestimmen der a priori Typen





Bestimmen der a posteriori Typen



Operatoridentifikation

- Operatoridentifikation ist signaturabhängige Namensanalyse für Funktionen
- a posteriori Typ der Operanden und damit die notwendigen Typanpassungen werden durch die Operatoridentifikation mit festgelegt