EIGENSCHAFTEN VON PROGRAMMIERSPRACHEN - DEFINIERT DURCH ATTRIBUTIERTE GRAMMATIKEN

Uwe Kastens
Institut für Informatik II
Universität Karlsruhe
Postfach 6380
7500 Karlsruhe 1

ZUSAMMENFASSUNG: Attributierte Grammatiken werden zur formalen Definition statischer Eigenschaften von Programmiersprachen und zur Spezifikation von Übersetzern angewandt. Für eine Reihe von wichtigen kontextabhängigen Spracheigenschaften (Gültigkeitsbereichsregeln, Regeln zur Typbestimmung und -anpassung) aus verschiedenen Sprachen geben wir Definitionen in Form von attributierten Grammatiken an.

ABSTRACT: Attributed grammars are both used for formal definitions of static properties of programming languages and for compiler specification. Attributed definitions of several important properties (scope rules, type determination and coercion) of different languages are shown.

1 Einführung

Attributierte Grammatiken (AG) eignen sich zur formalen Definition der statischen Eigenschaften von Programmiersprachen. Das Beschreibungsmittel erzwingt Vollständigkeit und Konsistenz der Definition, die weitgehend automatisch überprüft werden können [5]. Eine wohldefinierte AG [8], [5] dokumentiert die Abhängigkeiten von Eigenschaften der Sprachelemente, die durch den Kontext bestimmt werden, und schliesst zyklische Abhängigkeiten aus. Aus diesen Spezifikationen kann die Ablaufstruktur des Analyseteils von Übersetzern systematisch hergeleitet werden [5]. Setzt man zur Definition Beschreibungssprachen ein, die nicht nur Abhängigkeiten zwischen Attributen spezifizieren, sondern auch die Abstraktion von Spracheigenschaften durch typisierte Attributwerte beschreiben (wie in ALADIN [6] oder EAG [12]), so kann die Bestimmung aller

kontextabhängigen Eigenschaften vollständig in der AG in geschlossener Form spezifiziert werden. Eine solche Sprachdefinition ist zugleich ein vollständige formale Spezifikation des Analyseteils von Übersetzern. Deshalb wird dieses Beschreibungsmittel sowohl zur Sprachdefinition als auch im Übersetzerbau zunehmend eingesetzt (z. B. für die Sprachen LIS [11], PASCAL [12], PEARL [7]).

Die wichtigsten statischen Eigenschaften höherer Programmiersprachen sind die Aussagen über die Gültigkeit von Definitionen und über Typen von Sprachelementen. Unsere Erfahrung mit dem Entwurf von attributierten Grammatiken für verschiedene Sprachen haben gezeigt, dass man zur Beschreibung solcher Eigenschaften Standardtechniken einsetzen kann. In Abschnitt 2 diskutieren wir die Beschreibung verschiedener Gültigkeitsbereichsregeln, die für ALGOL 60, für ein-Pass-Übersetzbare Sprachen und für FORTRAN typisch sind. In Abschnitt 3 zeigen wir, wie einfache und komplexe Typen von Programmobjekten durch Attributwerte abstrakt beschrieben werden, und wie Aussagen über Typen (z. B. Typäquivalenz und Typanpassung) in einer AG definiert werden. Ausserdem werden Techniken zur zyklenfreien Beschreibung von Typdefinitionen und zyklisch definierten Typen demonstriert.

An dieser Stelle soll die Beschreibungsmethode, die den AG zugrunde liegt, nur kurz dargestellt werden. Einführende und präzisere Darstellungen findet man u. a. in [8], [14], [4], [5]. Eine AG enthält eine kontext-freie Grammatik, deren Symbolen Attribute zugeordnet sind. Jedes Attribut beschreibt eine kontextabhängige Eigenschaft des Symbols (z. B. den Typ eines Ausdrucks). Den syntaktischen Regeln sind semantische Regeln zugeordnet. Sie definieren jeweils einen Wert eines Attributs zu einem Symbol, das in der syntaktischen Regel auftritt, abhängig von anderen Attributen (z. B. der Typ einer indizierten Benennung ist der Elementtyp der Reihung).

Aus einem Strukturbaum für einen Satz der kontext-freien Grammatik erhält man einen attributierten Strukturbaum, indem man an jedem Baumknoten (der ein Exemplar eines Symbols der Ableitung repräsentiert) für jedes Attribut des Symbols einen Attributwert ergänzt. Dieser Wert repräsentiert eine Eigenschaft des Symbols in dem konkreten Kontext (z. B. der Ausdruck hat den Typ 'int'). Die zur Ableitung des Symbols angewandten syntaktischen Regeln entscheiden, durch welche semantische Regeln der Attributwert bestimmt wird. Wir unterscheiden erworbene Attribute (Eigenschaften), deren Werte durch den äusseren Kontext des Symbols bestimmt werden und abgeleitete Attribute, deren Werte durch die aus dem Symbol abgeleiteten Programmelemente bestimmt werden.

Den syntaktischen Regeln sind ausserdem Kontextbedingungen in Form

von Aussagen über Attributwerte zugeordnet. Sie schränken die Satzmenge der kontext-freien Grammatik auf die Sprache ein, die die AG definiert. Zu jeder Anwendung einer syntaktischen Regel in der Ableitung eines Satzes müssen die zugeordneten Kontextbedingungen erfüllt sein.

Eine AG wird zweckmässig in folgenden Schritten entworfen:

a) Entwurf der kontext-freien Syntax. Eine schon vorliegende Grammatik kann häufig vereinfacht werden, indem man Einschränkungen kontextabhängig beschreibt (siehe Abschnitt 3).

b) Festlegen des Wertebereichs von Attributen, die Eigenschaften von Programmobjekten (Typ, Zugriffsrechte, usw.) beschreiben.

c) Entwurf der semantischen Regeln.

d) Formulierung von Funktionen zur Attributberechnung (z. B. Typabgleich, Typäquivalenz).

2 Gültigkeitsbereiche und Bezeichner-Identifikation

Eine grundlegende gemeinsame Eigenschaft der meisten höheren und vieler niederer Programmiersprachen ist die Benennung von Programmobjekten durch Bezeichner, die in Definitionen eingeführt und an die Objekte gebunden werden (z. B. Variablendefinition). Diese Bindung ist in den meisten Sprachen statisch, d. h. sie kann festgestellt werden, ohne das Programm auszuführen. (Ausnahmen sind Sprachen mit vorwiegend interpretativem Charakter, wie LISP, SNOBOL, APL.)

Wir nennen das Auftreten eines Bezeichners in einem Sprachelement, das ihm ein Objekt (oder allgemeiner: eine Bedeutung) zuordnet, definierend. Das Auftreten eines Bezeichners heisst angewandt, wenn auf die ihm zugeordneten Eigenschaften Bezug genommen wird. Die Bezeichner-Identifikation ist die Zuordnung des angewandten Auftretens eines Bezeichners zu einem definierenden Auftreten - und damit zu den ihm zugeordneten Eigenschaften. Der Teil eines Programms, in dem angewandte Auftreten eines Bezeichners ein bestimmtes definierendes Auftreten identifizieren, heisst Gültigkeitsbereich der Definition.

Die Möglichkeiten zur hierarchischen Strukturierung von Programmen einer Sprache bestimmen auch die Regeln für die Gültigkeitsbereiche von Definitionen: In Block-strukturierten Sprachen sind die Gültigkeitsbereiche von Bezeichnern geschachtelt. Sprachen mit

einer beschränkten Zahl von Hierarchiestufen (wie FORTRAN oder COBOL) lassen Gültigkeitsbereiche nur auf diesen Ebenen zu. Weitere Unterschiede bezüglich der Gültigkeitsbereichsregeln ergeben sich aus der Möglichkeit, Bezeichner implizit durch ihre Anwendung zu definieren (FORTRAN), der Zulässigkeit (ALGOL 60) oder dem Verbot (LIS) von Definitionen des gleichen Bezeichners, die einander verdecken, oder dem Ziel, eine Sprache so zu definieren, dass sie ein-Pass-übersetzbar ist (PASCAL).

In einer AG beschreibt man Eigenschaften, die mit Gültigkeitsregeln zusammenhängen nach folgendem Prinzip: Eine Definition ist ein Paar (Bezeichner, Beschreibung des zugeordneten Objekts). Jedes Sprachelement, das Bezeichner enthalten kann (z. B. Ausdruck), liegt im Gültigkeitsbereich einer Menge von Definitionen. Diese wird als Attribut (in den Beispielen: at_Umgebung) allen solchen Sprachelementen zugeordnet. Die Regeln zur Berechnung dieser Attribute beschreiben die Gültigkeitsbereichsregeln der Sprache. Wir unterscheiden semantische Regeln, die Mengen von Definitionen zu einer neuen Umgebung zusammenfassen, und solche, die den Wert des Umgebungs-Attributs unverändert durch den Programmbaum "transportieren". Die ersteren werden den Sprachelementen zugeordnet, die Gültigkeitsbereiche begrenzen (z. B. Block). Wir zeigen unten, wie man abkürzend auf die letzteren verzichten kann. Die Bezeichner-Identifikation reduziert sich auf eine semantische Regel, zu jedem angewandten Auftreten, welche die Definition des Bezeichners im Umgebungsattribut aufsucht.

```
r1a:  RULE   Benennung ::= Bezeichner
      STATIC Benennung.at_Type := f_identifiziere
                (Bezeichner.at_Bez, Benennung.at_Umgebung)
      END
```

Die Funktion f_identifiziere liefert die Objektbeschreibung (z. B. den Typ) der für den Bezeichner gültigen Definition.

Für Sprachen mit Gültigkeitsregeln nach dem Muster von ALGOL 60 [10] gilt:

> Eine Definition ist im kleinsten sie umfassenden Block gültig, ausgenommen innenliegende Blöcke, die eine Definition des gleichen Bezeichners enthalten (Verdeckungsregel).

Diese Spracheigenschaft wird durch folgende Regel beschrieben:

```
r2a:  RULE   Block ::= 'BEGIN' Definitionen Anweisungen 'END'
      STATIC Block.at_lokale_Def :=
               Definitionen.at_Def + Anweisungen.at_Marken_Def
             Block.at_Umgebung :=
               Block.at_lokale_Def + Block.at_äussere_Umgebung;
             Definitionen.at_Umgebung := Block.at_Umgebung;
             Anweisungen.at_Umgebung := Block.at_Umgebung;
             CONDITION f_eindeutig (Block.at_lokale_Def)
      END
```

Der Operator + im obigen Kontext vereinigt Mengen von Definitionen. Wir gehen davon aus, dass sie durch endliche Folgen von Paaren (Bezeichner, Typ) repräsentiert werden. Die Vereinigung bedeutet dann Konkatenation der Folgen. Die Verdeckungsregel wird durch die Funktion f_identifiziere beschrieben: Sie wählt die erste Definition zu dem gesuchten Bezeichner aus der Folge von Definitionen aus. Die Kontextbedingung CONDITION... stellt sicher, dass es in einem Block keine zwei Definitionen zum gleichen Bezeichner gibt.

Abbildung 2.1 zeigt Attributabhängigkeiten in einem nach diesen Regeln attributierten Strukturbaum. Zyklische Abhängigkeiten treten nicht auf, da angewandte Auftreten von Bezeichnern in Definitionen nicht zu den statischen Eigenschaften des definierten Objekts beitragen.

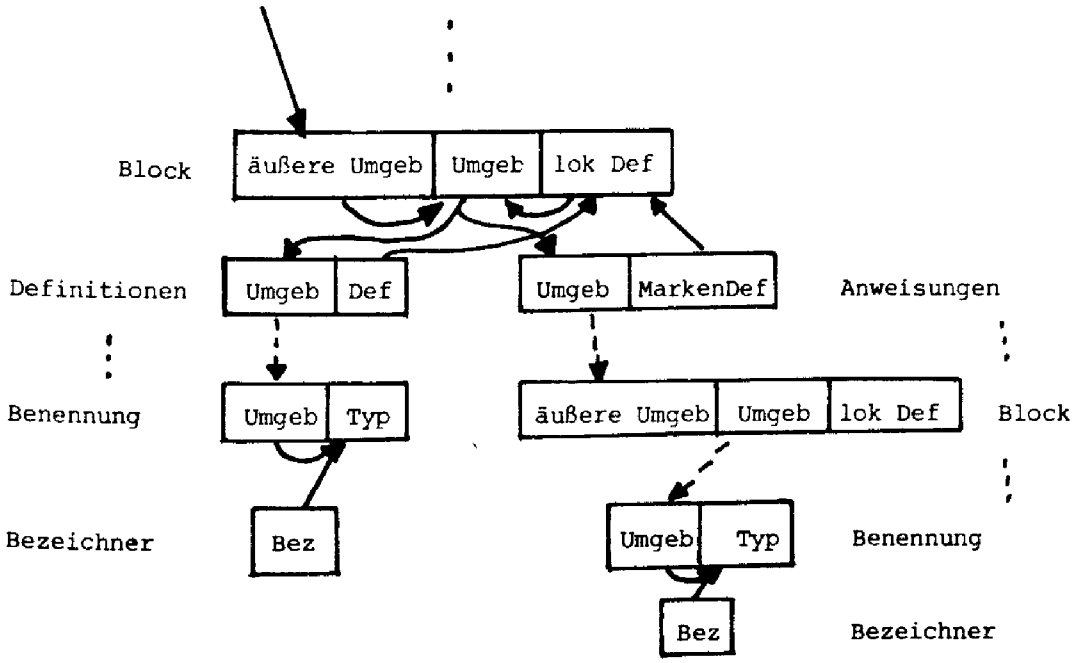

Abbildung 2.1
Gültigkeitsbereiche für ALGOL 60

Alle Attribute at_Umgebung von Baumknoten, die Söhne des gleichen

Block-Knotens sind, haben den gleichen Wert. Sie dienen nur dazu, die Umgebung an die Anwendungsstellen von Bezeichnern zu transportieren. Zur Verbesserung der Übersichtlichkeit führen wir dafür eine Abkürzung ein: Mit der Konstruktion

INCLUDING Block.at_Umgebung

wird auf das Attribut at_Umgebung des nächst äusseren Block-Knotens Bezug genommen. Eine solche Vereinfachung ist natürlich; denn offensichtlich hat nur der Block die Eigenschaft, eine neue lokale Umgebung zu definieren, und für die angewandten Auftreten von Bezeichnern reicht es aus festzustellen, welches der kleinste sie umfassende Block ist. Damit vereinfachen sich die Regeln r1a und r2a zu

```
r1b:  RULE   Benennung ::= Bezeichner
      STATIC Benennung.at_Typ := f_identifiziere
               (Bezeichner.at_Bez, INCLUDING Block.at_Umgebung)
END

r2b:  RULE   Block ::= 'BEGIN' Definitionen Anweisungen 'END'
      STATIC Block.at_lokale_Def :=
               Definitionen.at_Def + Anweisungen.at_Marken_Def;
             Block.at_Umgebung :=
               Block.at_lokale_Def + INCLUDING Block.at_Umgebung;
             CONDITION f_eindeutig (Block.at_lokale_Def)
      END
```

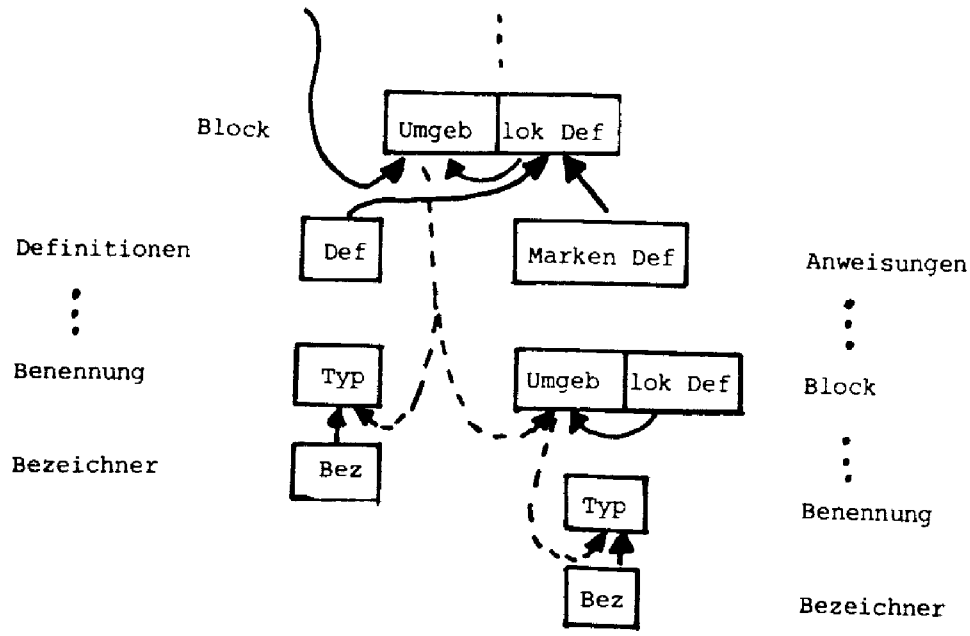

Abbildung 2.2
Gültigkeitsbereiche für ALGOL 60 (vereinfacht)

Die Attributabhängigkeiten in einem attributierten Strukturbaum zeigt Abbildung 2.2.

Sprachen, deren Definition auf die ein-Pass-Übersetzbarkeit ausgerichtet ist, verlangen, dass zu jedem Bezeichner das definierende vor dem angewandten Auftreten steht. Typisches Beispiel ist PASCAL [3], das im übrigen die ALGOL 60 Gültigkeitsregeln vorschreibt. Anders als in ALGOL 60 beginnt hier der Gültigkeitsbereich einer Definition nicht am Anfang eines Blocks, sondern unmittelbar nach der Definition. Jede Definition führt deshalb eine neue Umgebung ein, die um die vorangehende Definition erweitert ist:

```
r3: RULE   Block ::= 'BEGIN' Definitionen Anweisungen 'END'
    STATIC Block.at_Umgebung :=
             Definitionen.at_Def + Block.at_äussere_Umgebung;
           CONDITION f_eindeutig (Definitionen.at_Def)
    END;

r4: RULE   Definitionen ::= Definition
    STATIC Definiton.at_Umgebung :=
             INCLUDING Block.at_äussere_Umgebung;
           Definitionen.at_Def := Definition.at_Def
    END;

r5: RULE   Definitionen ::= Definitionen ';' Definition
    STATIC Definition.at_Umgebung :=
             Definitionen[2].at_Def +
             INCLUDING Block.at_äussere_Umgebung;
           Definitionen[1].at_Def :=
             Definitionen[2].at_Def + Definition.at_Def
    END;
```

(Mehrfach in der syntaktischen Regel auftretende Symbole werden in den semantischen Regeln durch Indizierung unterschieden.)

```
r6: RULE   Benennung ::= Bezeichner
    STATIC Benennung.at_Typ :=
             f_identifiziere (Bezeichner.at_Bez,
                              INCLUDING (Block.at_Umgebung,
                                         Definition.at_Umgebung))
    END
```

Das angewandte Auftreten von Bezeichnern wird in der Umgebung der nächst äusseren Definition identifiziert, falls es innerhalb einer Definition liegt, oder in der Umgebung des nächst äusseren Blocks, falls es zu einer Anweisung gehört. Die Abhängigkeiten in einem attributierten Strukturbaum zeigt Abbildung 2.3:

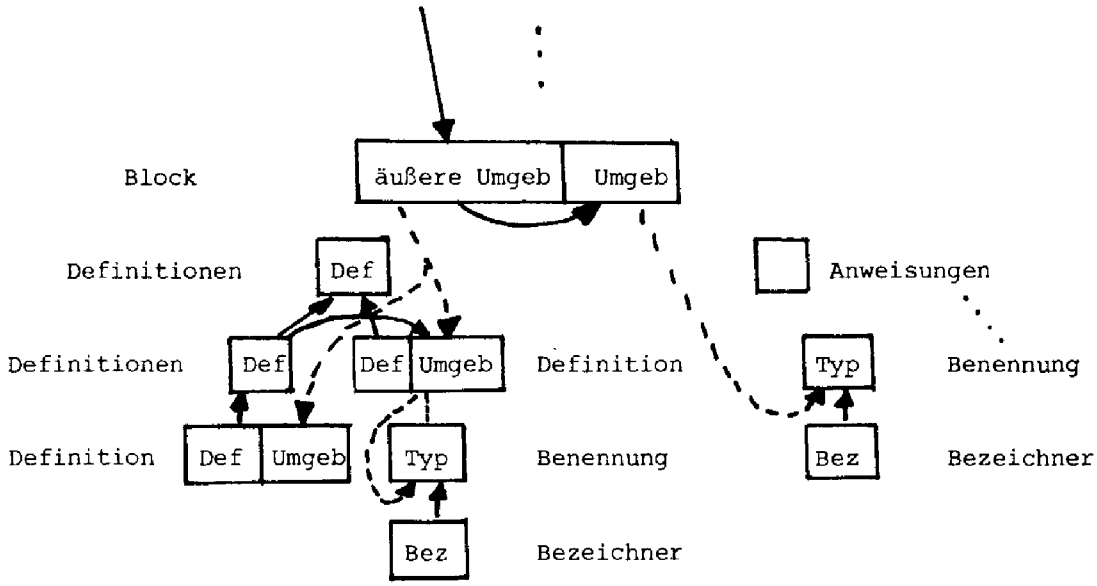

Abbildung 2.3
Gültigkeitsbereiche für PASCAL (vereinfacht)

Diese Attributierung definiert, dass der Gültigkeitsbereich einer Definition unmittelbar nach der Definition beginnt. Zusammen mit der Verdeckungsregel resultiert daraus, dass innerhalb eines einzigen Blocks ein angewandtes Auftreten eines Bezeichners entweder ein globales Objekt (vor einer lokalen Definition des Bezeichners) oder ein lokales Objekt identifizieren kann (nach einer lokalen Definition). Die Sprache Burroughs Extended Algol [2] hat diese Eigenschaft. Für PASCAL ist jedoch explizit verboten, dass vor einer Definition ein angewandtes Auftreten des gleichen Bezeichners im gleichen Block steht. Diese Eigenschaft kann man überprüfen, indem man die Menge der innerhalb von Definitionen angewandt auftretenden Bezeichner als Attributwert mitführt und gegen die nachfolgenden Definitionen prüft (ist im obigen Beispiel nicht enthalten.)

Ein-Pass-übersetzbare Sprachen enthalten häufig sogenannte "Vorwärtsdefinitionen", die es erlauben, indirekt rekursive Prozeduren zu definieren. Sie enthalten das definierende Auftreten des Prozedurbezeichners. Die nachfolgende Prozedurdefinition dazu erweitert die Umgebung nicht. Ein Attribut, das durch die Definitionen mitgeführt wird und das eine Menge der noch nicht gelösten Vorwärtsreferenzen enthält, stellt sicher, dass es zu jeder "Vorwärtsdefinition" genau eine Definition gibt. Ähnlich beschreibt man Markendefinitionen im Definitionsteil, zu denen es genau eine Markendefinition vor einer Anweisung geben muss. Auf die besonderen Probleme, die im Zusammenhang mit Typdefinitionen

entstehen, gehen wir in Abschnitt 3 ein.

Eine dritte Klasse der Gültigkeitsregeln finden wir in FORTRAN [1]: Jede separat übersetzbare Einheit (Haupt- oder Unterprogramm) begrenzt die Gültigkeit der darin definierten Bezeichner; mit folgenden Ausnahmen: Unterprogramm-Bezeichner und Bezeichner von "common blocks" werden als externe Referenzen aufgefasst, die der Binder löst. "Statement functions" begrenzen den Gültigkeitsbereich ihrer Parameter.) Gültigkeitsbereiche sind nicht geschachtelt. Bezeichner, die nicht explizit definiert sind, werden implizit durch ihr erstes angewandtes Auftreten eingeführt. Definitionen und Anweisungen können beliebig gemischt werden. Eine attributierte Grammatik beschreibt diese Eigenschaften wie folgt:

```
r7: RULE   Rumpf ::= Anweisungen
    STATIC Rumpf.at_expl_Def := Anweisungen.at_expl_Def;
           CONDITION f_eindeutig (Anweisungen.at_expl_Def)
    END;

r8: RULE   Anweisungen ::= Anweisung
    STATIC Anweisungen.at_expl_Def := Anweisung.at_expl_Def;
           Anweisung.at_Umgebung :=
               INCLUDING Rumpf.at_expl_Def;
           Anweisungen.at_impl_Def := Anweisung.at_impl_Def
    END;

r9: RULE   Anweisungen ::= Anweisungen Anweisung
    STATIC Anweisungen[1].at_expl_Def :=
               Anweisungen[2].at_expl_Def + Anweisung.at_expl_Def;
           Anweisung.at_Umgebung :=
               Anweisungen[2].at_impl_Def +
               INCLUDING Rumpf.at_expl_Def;
           Anweisungen[1].at_impl_Def :=
               Anweisungen[2].at_impl_Def + Anweisung.at_impl_Def
    END;

r10: RULE   Anweisung ::= Marke deklarative_Anweisung
     STATIC Anweisung.at_expl_Def :=
              Marke.at_Marken_Def + deklarative_anweisung.at_Def;
            Anweisung.at_impl_Def := { }
     END;
```

```
r11: RULE   Anweisung ::= Marke ausführbare_Anweisung
     STATIC Anweisung.at_expl_Def := Marke.at_Marken_Def;
            ausführbare_Anweisung.at_Umgebung :=
              Anweisung.at_Umgebung;
            Anweisung.at_impl_Def :=
                ausführbare_Anweisung.at_impl_Def
     END;
     ...
r12: RULE   Benennung ::= Bezeichner
     STATIC Benennung.at_impl_Def :=
              IF f_ist_in_Umgebung (Bezeichner.at_Bez,
                                    Benennung.at_Umgebung)
              THEN { }
              ELSE tp_Definition(Bezeichner.at_Bez,
                                 f_berechne_Typ (Bezeichner.at_Bez))
              FI;
            Benennung.at_Typ :=
              f_identifiziere (Bezeichner.at_Bez,
                               Benennung.at_impl_Def +
                                 Benennung.at_Umgebung)
     END
```

Anhand der graphischen Darstellung der Abhängigkeiten in einem attributierten Strukturbaum (Abbildung 2.4) wird deutlich, dass für die Bezeichner-Identifikation zwei Durchläufe nötig sind: Im ersten werden die expliziten Definitionen gesammelt, im zweiten wird identifiziert und gegebenenfalls die Menge der Definitionen erweitert.

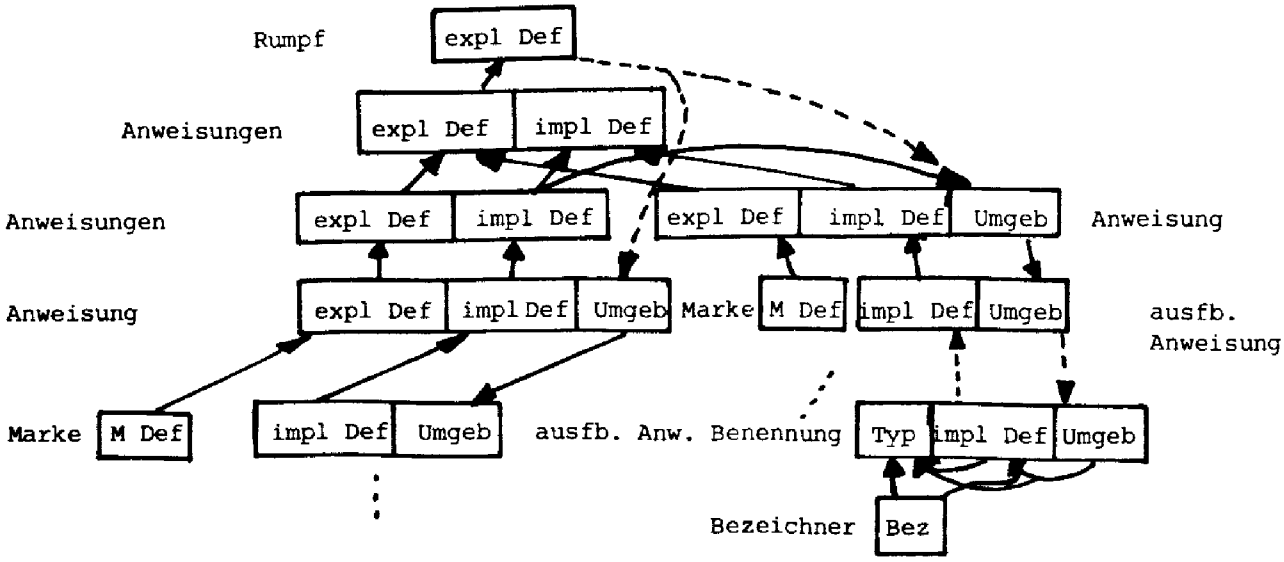

Abbildung 2.4
Gültigkeitsbereiche für FORTRAN

3 Typen

Der Typ eines Programm-Objektes bestimmt den Wertebereich und/oder die Operationen, die mit dem Objekt ausgeführt werden können. Insbesondere für Sprachen mit strikter Typbindung (ALGOL 68, PASCAL, SIMULA, usw.) haben die Aussagen über Typen von Objekten zentrale Bedeutung und zählen überwiegend zu den statisch überprüfbaren Eigenschaften der Programme. Wir diskutieren in diesem Abschnitt die Beschreibung solcher Eigenschaften durch attributierte Grammatiken für Sprachen mit komplexer Typstruktur (ALGOL 68, PASCAL). Die Beschreibung von Sprachen mit geringerer Typenvielfalt (ALGOL 60, FORTRAN) ergibt sich durch Vereinfachung daraus.

Zu den kontextabhängigen Eigenschaften vieler Elemente einer Sprache (z. B. Ausdrücke, Definitionen) zählt ihr Typ. Er wird durch ein Attribut des Programmelements beschrieben. Die Typenvielfalt der Sprache bestimmt den Wertebereich dieser Attribute. Wir unterscheiden die konkrete Syntax der Typangaben (z. B. in Definitionen) und die abstrakte Beschreibung von Typen durch Attributwerte. Im ersten Teil dieses Abschnitts stellen wir die Abbildung der konkreten Syntax auf Attributwerte dar, im zweiten untersuchen wir die speziellen Probleme, die im Zusammenhang mit Typdefinitionen auftreten.

3.1 Abstrakte Beschreibung von Typen

Die abstrakte Beschreibung eines Typs kann aus drei Gründen von der konkreten Syntax abweichen:

a) Die konkrete Syntax ist redundant.

b) Die konkrete Syntax kann Angaben enthalten, die nicht zu den statischen Eigenschaften des Typs zählen (z. B. dynamisch bestimmte Reihungsgrenzen)

c) Die konkrete Syntax beschränkt die Typenvielfalt in bestimmtem Kontext (z. B. für Typen von Parametern oder Reihungselementen).

Der letzte Aspekt gibt Anlass zu der Überlegung, ob solche Einschränkungen nicht besser durch kontextabhängige semantische Regeln zu einer vergröberten kontext-freien Syntax definiert werden. Diese Entscheidung, die natürlich nur für kontext-frei beschreibbare Einschränkungen ansteht, sollte man im wesentlichen aufgrund des Beschreibungsaufwands treffen: Er ist lokal betrachtet für die kontext-freie Beschreibung häufig kleiner, kann jedoch eine

sonst vermeidbare Gliederung der Typen in Klassen (z. B. elementare Typen, einfache Typen, zusammengesetzte Typen) oder eine teilweise Duplizierung der konkreten Syntax erfordern (z. B. formaler Typ, aktueller Typ, formaler Reihungstyp, aktueller Reihungstyp). Diese Aufwandsüberlegungen gelten entsprechend für den Übersetzerbauer, der entscheidet, ob solche Restriktionen durch die kontext-freie Zerteilung oder die semantische Analyse überprüft werden. Kann in einer Sprache an einer Stelle einer expliziten Artangabe auch ein Typbezeichner stehen, so können Restriktionen häufig erst nach Identifikation des Typbezeichners überprüft und deshalb nicht kontext-frei beschrieben werden.

Im folgenden betrachten wir Beschreibung von einfachen Typen, wie int, real, bool, Reihungstypen, Referenztypen und Verbundtypen. Der Wertebereich eines Attributs, das den Typ eines Sprachelements beschreibt, ist daher die Vereinigung der Abstraktionen dieser Typen beschrieben durch folgende Attributtypdefinition:

```
TYPE tp_type: UNION (tp_einfach, tp_Reihung,
                     tp_Referenz, tp_Verbund)
```

Zur abstrakten Beschreibung einfacher Typen reicht die Angabe der Typzugehörigkeit aus. Der Attributtyp ist ein Aufzählungstyp.

```
TYPE tp_einfach : (sc_int, sc_real, sc_bool);

r13: RULE   Typ ::= 'INTEGER'
     STATIC Typ.at_Typ := sc_int
     END;
     ...
```

Für Reihungstypen nehmen wir an, dass die Reihungsgrenzen nicht statische Eigenschaften des Typs sind.

```
TYPE tp_Reihung : STRUCT (Stufen          : INT,
                          formale_Grenzen : BOOL,
                          Element_Typ     : tp_Typ);

r14: RULE   Typ ::= '[' Grenzen ']' Typ
     STATIC Typ[1].at_Typ := tp_Reihung (Grenzen.at_Stufen,
                                         Grenzen.at_formal,
                                         Typ[2].at_Typ);
            CONDITION NOT (Typ[2].at_Typ IS tp_Reihung)
     END;

r15: RULE   Grenzen ::= aktuelle_Grenzen
     STATIC Grenzen.at_formal := FALSE;
            Grenzen.at_Stufen := aktuelle_Grenzen.at_Stufen
     END;
```

```
r16: RULE   Grenzen ::= formale_Grenzen
     STATIC Grenzen.at_formal := TRUE;
            Grenzen.at_Stufen := aktuelle_Grenzen_at.Stufen
     END
```

Müssen die Grenzen statisch berechenbar sein, und tragen sie zur Unterscheidung von Typen bei (wie in PASCAL), so wird die Abstraktion um entsprechende Komponenten erweitert. Die Kontextbedingung in r14 schliesst Reihungstypen als Elementtyp aus.

```
     TYPE tp_Referenz : STRUCT (Referierter_Typ : tp_Typ);

r17: RULE   Typ ::= 'REF' Typ
     STATIC Typ[1].at_Typ := tp_Referenz (Typ[2].at_Typ);
            CONDITION IF Typ[2].at_Typ IS tp_Reihung
                      THEN Typ[2].at_Typ.formale_Grenzen
                      ELSE TRUE
                      FI
     END
```

Die Kontextbedingung in r17 stellt sicher, dass der referierte Typ kein aktueller Reihungstyp ist.

Verbundtypen werden durch die Folge der Komponentendefinitionen beschrieben:

```
     TYPE tp_Verbund_Typ : LISTOF tp_Definition;
     TYPE tp_Definition :  STRUCT (Bez : STRING,
                                   Objekt_Typ : tp_Typ);

r18: RULE   Typ ::= 'STRUCT' '('  (Komponente //',') ')'
     STATIC Typ.at_Typ := tp_Verbund_Typ (Komponente.at_Def)
     END

r19: RULE   Komponente ::= Typ Bezeichner
     STATIC Komponente.at_Def :=
              tp_Definition (Bezeichner.at_Bez, Typ.at_Typ);
            CONDITION IF Typ_at.Typ IS tp_Reihungstyp
                      THEN NOT Typ.at_Typ.formale_Grenzen
                      ELSE TRUE FI
            END
```

Variablendefinitionen können z. B. durch folgende Regel beschrieben werden:

```
r20: RULE   Definition ::= Typ Bezeichner [':=' Ausdruck]
     STATIC Definition.at_Definition :=
              tp_Definition (Bezeichner.at_Bez,
                             tp_Referenz(Typ.at_Typ));
            CONDITION IF Typ.at_typ IS tp_Reihung
                      THEN NOT Typ.at_Typ.formale_Grenzen
                      ELSE TRUE FI;
            Ausdruck.at_Typ_nach := Typ.at_Typ
     END
```

Die erste semantische Regel drückt aus, dass die Variablendefinition - wie in ALGOL 68 - ein Bezugsobjekt einführt, das Objekte des angegebenen Typs als Werte aufnehmen kann.

Die kontextabhängigen Aussagen über Typen in einer Sprachdefinition lassen sich auf einige Grundmuster zurückführen, die in der attributierten Grammatik durch rekursive Funktionen, über den abstrakten Typbeschreibungen (Werten des Wertebereichs von tp_Typ) formuliert werden.

1. Typgleichheit wird durch die Gleichheit der Abstraktion beschrieben, falls es in der Sprache keine Typbezeichner als abkürzende Schreibweise für Typen gibt. In diesem Fall müssen in einer Typabstraktion enthaltene Typbezeichner identifiziert und expandiert werden. Ausserdem muss die Terminierung der Funktion auch für zyklisch definierte Typen sichergestellt sein.

2. Typ Verträglichkeit ist im allgemeinen eine nicht symmetrische Relation zwischen Typen: In unserem Beispiel ist ein aktueller Reihungstyp mit einem formalen verträglich, wenn Elementtyp und Anzahl der Stufen übereinstimmen. In Sprachen, die Zugriffsrechte (variabel oder konstant) zum Typ hinzurechnen (z. B. LIS, BALG, PEARL), werden sie bei der Typ_Verträglichkeit berücksichtigt.

3. Typ Anpassung: Im allgemeinen gilt die Regel, dass ein Typ t1 an einen Typ t2 anpassbar ist, wenn t1 mit t2 verträglich ist oder t1 durch eine Folge von Anpassungsoperationen, wie Weiten (Übergang von sc_int nach sc_real) oder Dereferenzieren (Übergang von tp_Referenz nach dem referierten Typ) in einen mit t2 verträglichen Typ überführt werden kann.

4. Typ Abgleich ist im allgemeinen so definiert, dass zu einer Folge von Ausgangstypen ein Zieltyp bestimmt wird, so dass jeder Ausgangstyp an den Zieltyp anpassbar ist, und die Anzahl der notwendigen Anpassungsoperationen minimal ist.

Wir verzichten hier auf eine inhaltliche Diskussion dieser

Funktionen, da ihre Formulierung nicht für attributierte Grammatiken typisch ist.

Ihre Anwendung auf Attribute von Sprachelementen soll am Beispiel der Typ_Anpassung erläutert werden. Die folgenden Überlegungen gelten für Sprachen, die implizite Typanpassungen zulassen. In diesen Sprachen ordnet man jedem (Teil-) Ausdruck zwei Attribute zu: den Typ vor Anwendung von impliziten Anpassungsoperationen (at_Typ_vor), der unabhängig vom Kontext des Ausdrucks bestimmt wird - sieht man einmal von der Bezeichner-Identifikation ab - und den Typ, den der Ausdruck nach Anwenden von Anpassungsoperationen annehmen muss, um den Kontextbedingungen zu genügen (at_Typ_nach). In r20 definiert die letzte semantische Regel, dass der Initialisierungsausdruck an den in der Definition angegebenen Typ anzupassen ist. Der Ausdruck steht in einer "starken" syntaktischen Position (in ALGOL 68-Terminologie); denn der Typ nach der Anpassung wird unabhängig vom Typ vor der Anpassung bestimmt. Der Ausdruck in der folgenden Regel muss auch ggf. angepasst werden; jedoch an einen Typ, der vom ursprünglichen Typ abhängt.

```
r21: RULE   Selektion ::= Ausdruck '.' Zeiger
     STATIC Ausdruck.at_Typ_nach :=
              f_passe_an_Verbund_an (Ausdruck.at_Typ_vor);
            Selektion.at_Typ_vor :=
              f_identifiziere_Komponente
                (Zeiger_Bez, Ausdruck.at_Typ_nach);
            Selektion.at_Anpassungssequenz :=
              f_passe_an (Selektion.at_Typ_nach,
                          Selektion.at_Typ_vor);
            CONDITION Selektion.at_Anpassungssequenz ≠
                      Anpassung_undefiniert
     END
```

Die Regel r21 gibt ausserdem an, wie durch ein weiteres Attribut die auf Ausdrücke anzuwendende Folge von Anpassungsoperationen beschrieben wird.

3.2 Typdefinitionen

Viele moderne höhere Programmiersprachen erlauben es, in Typdefinitionen Bezeichner einzuführen, die einen Typ repräsentieren (SIMULA, ALGOL 68, PASCAL, LIS, PEARL). Die Definition dieser Spracheigenschaften durch eine attributierte Grammatik muss zwei Probleme lösen:

a) Die Identifikation von Typbezeichnern darf nicht zu zyklischen Attributabhängigkeiten führen.

b) Auch rekursive definierte Typen müssen durch endlich angebbare Attributwerte beschrieben werden.

Unabhängig vom Beschreibungsmittel führt die Interpretation von Typbezeichnern als abkürzende Schreibweise für einen Typ (ALGOL 68 [13], PEARL [7]) zu einer wesentlich komplizierteren Definition der Typgleichheit (siehe z. B. [9]). Dieses Problem wird in SIMULA und LIS dadurch vermieden, dass eine Typdefinition grundsätzlich einen neuen Typ einführt, der von allen anderen verschieden ist.

Das Problem der zyklischen Attributabhängigkeiten wird deutlich, wenn man in den Regeln r3 bis r6 Typdefinitionen unter die Definitionen einreiht, und Typbezeichner wie Benennungen im Umgebungs-Attribut identifiziert. Dieses Attribut enthält Informationen über Typen definierter Objekte, die wiederum von den identifizierten Typbezeichnern abhängen.

Wir lösen dieses Problem in drei Schritten:

1. Schritt: Die Identifikation der Typbezeichner wird "verzögert". Zunächst werden in die abstrakte Beschreibung von Typen die enthaltenen Typbezeichner selbst aufgenommen. Sie werden noch nicht durch die Abstraktion des Typs, den sie abkürzen, ersetzt. (Bei rekursiv definierten Typen ist eine vollständige Expansion ohnehin nicht möglich.)

2. Schritt: Objekte, deren Typ durch Typbezeichner beschrieben wird, können an Programmstellen zugänglich sein, an denen die zugehörigen Typdefinitionen nicht gültig sind, da sie durch andere Definitionen verdeckt sind. Es muss deshalb sichergestellt werden, dass ein Typbezeichner innerhalb der Menge von Definitionen identifiziert wird, die an der Stelle des angewandten Auftretens des Typbezeichners gilt; und nicht in der Menge, die dort gilt, wo der Typbezeichner innerhalb eines Attributwerts auftritt. Dies wird erreicht, indem man als Abstraktion für Typbezeichner ein Paar (Bezeichner, Umgebung) wählt. Die zweite Komponente ist die Menge der an der Anwendungsstelle gültigen Definitionen. Zu jedem Block wird wie in Schritt 1 beschrieben ein Attribut lokale_Def berechnet, wobei die Umgebungs-Komponente darin enthaltener Abstraktionen von Typbezeichnern leer ist. Daraus wird ein zweites Attribut Umgebung berechnet, in dem die zweite Komponente aller enthaltenen Abstraktionen von Typbezeichnern (wie oben beschrieben) bestimmt wird.

3. Schritt: Trifft man bei der Untersuchung eines Typattributs auf die Abstraktion eines Typbezeichners (z. B. beim Dereferenzieren des Typs REF t), so identifiziert man in der Umgebungs-Komponente die zugehörige Definition, ersetzt den Typbezeichner durch den Typ,

den er abkürzt, und ergänzt in allen darin enthaltenen Abstraktionen von Typbezeichnern die ursprüngliche Umgebung.

Dieses Verfahren entspricht der Technik, mit der ein Übersetzer das Problem löst: Die Umgebungs-Attribute entsprechen der Definitionstabelle. Typbezeichner werden nicht sofort identifiziert, sondern in einem separaten Durchgang durch die Tabelle durch Bezüge auf Definitionen ersetzt. Diese Bezüge entsprechen den Paaren (Bezeichner, Umgebung) in der attributierten Grammatik, die als statisches Beschreibungsmittel kein Bezugs- oder Variablenkonzept zulässt. Das vorgestellte Verfahren erlaubt auch die Beschreibung von Moduln, die lokale Objekte mit lokal definierten Typen exportieren (SIMULA, PEARL, LIS).

```
r24: RULE   Block ::= 'BEGIN' Definitionen Anweisungen 'END'
     STATIC Block.at_lokale_Def :=
              Definitionen.at_Def + Anweisungen.at_Marken_Def;
            Block.at_Umgebung :=
              f_vervollständige_Typbezeichner
                (Block.at_lokale_Grössen,
                 INCLUDING Block.at_Umgebung);
            CONDITION f_eindeutig (Block.at_lokale_Grössen)
     END;

     TYPE tp_Typbezeichner : STRUCT (Bez : STRING,
                                     Umgebung : tp_Definitionen);

r25: RULE   Typbezeichner ::= Bezeichner
     STATIC Typbezeichner.at_type :=
              tp_Typbezeichner (Bezeichner.at_Bez,
                                tp_Definitionen() )
                                (* vorläufig leere Folge *)
     END
```

Literatur

[1] American National Standard Programming language FORTRAN. American National Standard Institute, X 3.9, 1966 u. 1978

[2] Burroughs B6700/7700 ALGOL Language, Reference Manual Burroughs Corporation, Form No. 5000649, 1974

[3] Jensen, K., Wirth, N.: Pascal User Manual and Report. Springer Verlag Heidelberg 1974

[4] Kastens, U.: Einführung in attributierte Grammatiken. In: Fachgespräche Compiler-Compiler, Berlin 1978, Bericht der Techn. Hochschule Darmstadt, Fachbereich Informatik, 1978

[5] Kastens, U.: Ordered Attributed Grammars. Interner Bericht 7/78, Fakultät für Informatik, Universität Karlsruhe

[6] Kastens, U.: ALADIN - eine Beschreibungssprache auf der Basis attributierter Grammatiken. Interner Bericht 7/79, Fakultät für Informatik, Universität Karlsruhe

[7] Kastens, U., Köllner, R., Zimmermann, E.: Eine attributierte Grammatik für PEARL. Interner Arbeitsbericht, Fakultät für Informatik, Universität Karlsruhe, 1979

[8] Knuth, D.E.: Semantics of Context-free Languages. In: Math. Syst. Th. 2, 2, 1968 und 5, 1, 1971

[9] Koster, C.H.A.: On Infinite Modes. In: Algol Bulletin 30,3, 1969

[10] Naur, P. (ed.): Report on the algorithmic language ALGOL 60. In: Num. Math 2, 1960

[11] Schauer, J.: Eine attributierte Grammatik für LIS. Interner Arbeitsbericht, Fakultät für Informatik, Universität Karlsruhe

[12] Watt, D.A.: An Extended Attribute Grammar for PASCAL. SIGPLAN Notices 14, 2, 1979

[13] v. Wijngaarden, A. (ed.): Revised report on the algorithmic language ALGOL 68. In: Acta Informatica 5, 1975

[14] Wilhelm, R.: Attributierte Grammatiken. In: Informatik-Spektrum 2, 3, 1979