$FIRST_1$- und $FOLLOW_1$-Mengen nichtiterativ als Lösung eines reinen Vereinigungsproblems berechnet. Das beschriebene Verfahren zur Fehlerbehandlung in RLL(1)-Parsern ist eine Verfeinerung des von Ammann im Züricher Pascal-P4-Übersetzer realisierten Verfahrens [Amm78], [Wir78]. Sie ist beschrieben in [LDHS82].

LR(k)-Grammatiken wurden von Knuth [Knu65] eingeführt. Die für die Praxis wichtigen Teilklassen SLR(k) und LALR(k) wurden von DeRemer entdeckt [DeR69], [DeR71]. Eine sehr effiziente Berechnung von LALR(1)-Vorausschaumengen über die Formulierung als reines Vereinigungsproblem wurde von DeRemer und Pennello beschrieben [DP82].

Das vorgestellte Fehlerbehandlungsverfahren für LR(k)-Parser folgt [PD78].

Scannergenerierung mit LR-Techniken geht auf [DeR74] zurück.

In [CH87] werden verschiedene Syntaxanalyseverfahren mit und ohne Zurücksetzen in Form von Prolog-Programmen vorgestellt.

Kapitel 9

Semantische Analyse

9.1 Aufgabe der semantischen Analyse

Einige notwendige Eigenschaften von Programmen sind nicht durch eine kontextfreie Grammatik beschreibbar. Diese Eigenschaften werden durch Prädikate auf Kontextinformation, sogenannte Kontextbedingungen, beschrieben. Dazu gehören die Deklariertheitseigenschaften und die Typkonsistenz. Beide hängen von den Gültigkeits- und Sichtbarkeitsregeln der Programmiersprache ab.

Die **Gültigkeitsregeln** legen für im Programm deklarierte Bezeichner (Identifier) fest, in welchem Teil des Programms ihre Deklaration einen Effekt hat. Die **Sichtbarkeitsregeln** wiederum bestimmen, wo in seinem Gültigkeitsbereich ein Bezeichner sichtbar bzw. verdeckt ist.

Die **Deklariertheitseigenschaften** bestimmen etwa, daß zu jedem angewandt auftretenden Bezeichner eine explizite Deklaration gegeben werden muß und daß Doppeldeklarationen verboten sind.

Die **Typkonsistenz** eines Programms garantiert, daß zur Ausführungszeit keine Operation (außer Eingabeoperationen) auf Operanden angewendet wird, auf die sie von ihren Argumenttypen her nicht paßt.

Statische semantische Eigenschaften

Man bezeichnet eine (nicht kontextfreie) Eigenschaft eines Konstrukts einer Programmiersprache als eine **statische semantische** Eigenschaft, wenn

(1) für jedes Vorkommen dieses Konstrukts in einem Programm der „Wert“ dieser Eigenschaft für alle (dynamischen) Ausführungen des Konstrukts gilt (der Typ eines Ausdrucks, d.h. der „Wert der Typeigenschaft“, gibt den Typ – evtl. nur den allgemeinsten – aller Werte an, die sich dynamisch bei Auswertungen des Ausdrucks ergeben können), und wenn

(2) für jedes Vorkommen des Konstrukts in einem korrekten Programm diese Eigenschaft berechnet werden kann. (Sowohl in stark als auch in polymorph getypten Sprachen ist für jeden Ausdruck in einem korrekten Programm der bzw. ein Typ berechenbar.)

Die erste Bedingung beschreibt die Beziehung zwischen statischer und dynamischer Semantik. Dynamische semantische Eigenschaften sind i.a. erst zur Laufzeit von Programmen bekannt. Statische semantische Eigenschaften beschreiben

allen dynamischen Ausführungen gemeinsame Eigenschaften, die zur Übersetzungszeit berechnet werden können. Bedingung (1) deckt auch noch solche Eigenschaften ab, die man üblicherweise durch abstrakte Interpretation (Datenflußanalyse) berechnet. Die zweite Bedingung schließt solche Eigenschaften aus, da abstrakte Interpretation nur versucht, möglichst gute statische Annäherungen an dynamische Eigenschaften von Programmkonstrukten zu berechnen, und dabei die „leere" Information i.a. eine mögliche Information ist.

Betrachten wir etwa die Eigenschaft „Wert von Variablen", d.h. für jede Anweisung in einem Programm die Information, welchen Wert jede Programmvariable vor jeder Ausführung dieser Anweisung hat. Diese Information ist natürlich eine dynamische Eigenschaft. Sie läßt sich jedoch näherungsweise berechnen. Eine Annäherung an diese dynamische Eigenschaft besteht darin, daß nur für einige der Programmvariablen bei einigen Anweisungen der Wert berechnet werden kann.

Wird etwa die Information „Variable x hat den Wert 5" an einer Anweisung berechnet, so gilt dies für alle Ausführungen des Programms und dieser Anweisung. Insofern ist Bedingung (1) erfüllt. Andererseits ist eine mögliche Information bei einer Anweisung: „von keiner Programmvariablen ist hier bekannt, welchen Wert sie bei jedem Eintritt in diese Anweisung hat". Deshalb werden wir die Eigenschaft „Wert von Variablen" nicht als statische semantische Eigenschaft bezeichnen. Solche und ähnliche Eigenschaften werden durch abstrakte Interpretation berechnet.

Als weiteres Beispiel betrachten wir die Berechnung der Typ-Eigenschaft für arithmetische Ausdrücke. Wir nehmen an, daß wir von allen terminalen Operanden, Variablen und Konstanten, den Typ bereits kennen, und daß dieser entweder integer oder real ist. Dann kann man die Typberechnung für Ausdrücke, die mithilfe der Operatoren $+, -, *, /$ und $\div$ (ganzzahlige Division) gebildet werden, wie in Tabelle 9.1 angeben.

Tabelle 9.1: Typberechnung für einfache Operatoren

Operator	Typ, 1. Operand	Typ, 2. Operand	Typ Resultat
$+, -, *$	*int*	*int*	*int*
	int	*real*	*real*
	real	*int*	*real*
	real	*real*	*real*
$/$	*int* | *real*	*int* | *real*	*real*
$\div$	*int*	*int*	*int*

Unter der oben gemachten Annahme erfüllt die so beschriebene Typ-Eigenschaft die beiden Bedingungen für statische semantische Eigenschaften. Wenn sich nämlich für einen Ausdruck e der Typ t ergibt, so ist sichergestellt, daß bei jeder fehlerfreien Ausführung des e enthaltenden Programms die Auswertung von e einen Wert vom Typ t ergibt. Der einzige kritische Fall hier ist die zur Laufzeit mögliche Division durch 0. Enthalten die zugrundeliegenden Bereiche

der integer- und real-Zahlen das undefinierte Element des jeweiligen Typs, so liegt bei Division durch 0 als Ergebnis das undefinierte Element des Bereichs der real-Zahlen als Ergebnis des Ausdrucks vom Typ real vor. Außerdem läßt sich, wie Bedingung (2) es fordert, für jedes korrekte Programm der Typ jedes Ausdrucks berechnen. Im Falle eines Typfehlers ist die durch Tabelle 9.1 definierte Funktion auf einer Kombination von Argumenten nicht definiert.

Halten wir also fest, daß die Typ-Eigenschaft für Ausdrücke mit den oben eingeführten Operatoren eine statische semantische Eigenschaft ist. Es können auch noch einige übliche Operatoren hinzukommen, wie etwa das unäre Minus, die modulo-Funktion, der Absolutbetrag oder das Auf- bzw. Abrunden von reellen Zahlen zur nächsten ganzen Zahl, ohne daß es Schwierigkeiten gibt.

Jetzt wollen wir aber eine Erweiterung um den Potenz-Operator vornehmen und die Typberechnung von Potenzausdrücken wie in Algol60 definieren, siehe Tabelle 9.2.

Tabelle 9.2: Typberechnung für den Potenz-Operator

Operator	Typ, Größe 1. Operand	Typ, Größe 2. Operand	Typ Resultat
$\uparrow$	*int* | *real*	$int > 0$	*int* | *real* wie 1. Op.
	int | $real \neq 0$	$int = 0$	*int* | *real* wie 1. Op.
	int | $real \neq 0$	$int < 0$	*real*
	int | $real > 0$	*real*	*real*
	int | $real = 0$	$real > 0.0$	$real = 0.0$

Jetzt hängt der Typ eines Ausdrucks $e_1 \uparrow e_2$ nicht nur von den Typen von e_1 und e_2, sondern auch noch von der Größe des Werts von e_2 ab. Diese Größe ist i.a. keine statische Eigenschaft, sondern erst zur Laufzeit verfügbar. Haben e_1 und e_2 den Typ integer, so können wir also i.a. nicht statisch bestimmen, ob das Ergebnis der Auswertungen von $e_1 \uparrow e_2$ vom Typ integer oder real sein wird. Deshalb ist für die um Potenzierung erweiterten arithmetischen Ausdrücke die Typ-Eigenschaft keine statische semantische Eigenschaft mehr. Ein Übersetzer wird also Instruktionen erzeugen, die zur Laufzeit überprüfen, ob die auf eine Potenzierung folgenden Operationen mit Operanden richtigen Typs arbeiten. Wie im Fall der Division durch 0 sind hier auch Laufzeitfehler aufgrund nichtzugelassener Werte von e_1 und e_2 möglich.

Terminologie

Wir benutzen die folgenden Begriffe, um einige Aufgaben der semantischen Analyse zu beschreiben. Dabei listen wir hinter den Kürzeln (i), (f) und (l) jeweils Beispiele aus imperativen, funktionalen bzw. logischen Sprachen auf.

Ein **Bezeichner** (Identifier) ist ein Symbol (im Sinne der lexikalischen Analyse), welches in einem Programm zur Benennung eines Objektes benutzt werden kann. Objekte, die benannt werden können, sind Variablen in imperativen, funktionalen und logischen Sprachen, Konstante, Typen, Prozeduren, Funktionen, Prädikate und Funktoren.

Die **Deklaration** eines Bezeichners führt den Bezeichner als Benennung eines (meist in der Deklaration gegebenen) Objektes ein. Sein Vorkommen in einer Deklaration ist ein **definierendes Vorkommen**, alle anderen sind **angewandte Vorkommen**.

In jeder Programmiersprache gibt es Konstrukte, die die Gültigkeit von Bezeichnern begrenzen. Diese Konstrukte, etwa

(i) Prozedurdeklarationen, Blöcke, Pakete, Moduln,

(f) Funktionsdefinitionen, let-, letrec-, where-Konstrukte, und

(l) Klauseln

nennen wir **Scope-Konstrukte**. Vorkommen von Scope-Konstrukten in Programmen heißen **Blöcke**. Bei diesem Sprachgebrauch ist also eine Prozedurdeklaration in einem Pascal-Programm ein Block, ebenso wie eine Funktionsdefinition oder eine lokale Definition mit einem let-Ausdruck in einem funktionalen Programm oder eine Klausel in einem logischen Programm.

Der **Typ** eines Objekts gibt an, was während der Ausführung des zugehörigen Programms mit dem Objekt gemacht werden kann. Ein integer-Wert etwa kann durch dazu geeignete Operationen mit anderen Werten verknüpft werden, (i) an eine integer-Variable zugewiesen werden, (f) an eine integer-Variable gebunden werden. (f) Ein Objekt vom Typ $(t_1 \times t_2 \times \cdots \times t_n \to t)$ kann auf ein n-Tupel von Objekten der Typen $t_1, \ldots, t_n$ angewendet werden, um dann ein Ergebnis vom Typ t zu produzieren.

Konkrete und abstrakte Syntax

Die Eingabe für die semantische Analyse eines Programms ist ein durch die Syntaxanalyse erstellter Baum. Dieser Baum kann die konkrete oder die abstrakte Syntax des Programms darstellen. Die **konkrete Syntax** des Programms wird durch den Syntaxbaum gemäß der die Sprache definierenden kontextfreien Grammatik dargestellt. Die kontextfreie Grammatik zu einer Sprache enthält viele Informationen, die für die weitere Verarbeitung von Programmen nicht wichtig sind. Dazu gehören alle Terminalsymbole, die zwar für die syntaktische Analyse und für das Lesen von Programmen wichtig sind, aber keine semantische Bedeutung tragen, z.B. alle Schlüsselwörter. Außerdem drücken sich Präzedenzen von Operatoren in Schachtelungen von Nichtterminalen aus, ein Nichtterminal pro Präzedenztiefe. Diese Nichtterminale und zugehörigen Produktionen sind nicht mehr von Relevanz, wenn die syntaktische Struktur erkannt ist.

Deshalb benutzen Übersetzer zur expliziten Darstellung der syntaktischen Struktur von Programmen die **abstrakte Syntax**. Sie enthält lediglich den wesentlichen Teil der syntaktischen Struktur des Programms. Sie identifiziert nur noch die im Programm auftretenden Konstrukte und ihre Schachtelungsbeziehung.

Beispiel 9.1.1
Das Programmstück
if $x + 1 > y$
then $z := 1$
else $z := 2$
fi
hat (bei entsprechend hinzugedachter kontextfreier Grammatik) Bäume zur konkreten wie abstrakten Syntax, wie in Abbildung 9.1 und Abbildung 9.2 dargestellt. □

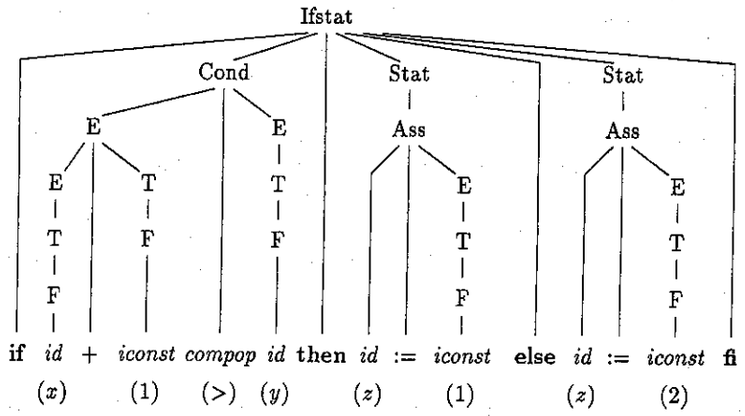

Abb. 9.1: Baum zur konkreten Syntax

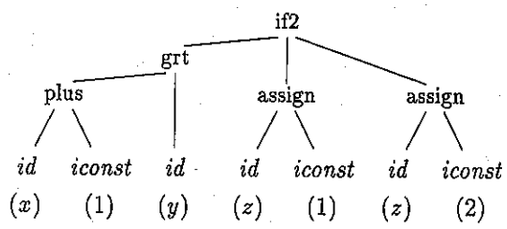

Abb. 9.2: Baum zur abstrakten Syntax

Im folgenden werden wir, je nachdem was vorteilhafter ist, die konkrete bzw. die abstrakte Syntax benutzen. Wir halten jedoch fest, daß Übersetzer immer die abstrakte Syntax zur semantischen Analyse und zur Codeerzeugung verwenden.

9.1.1 Gültigkeits- und Sichtbarkeitsregeln

Da Programmiersprachen es meist zulassen, daß mehrere Deklarationen des gleichen Bezeichners zur Bezeichnung verschiedener Objekte in einem Programm gegeben werden dürfen, bedarf es einer Regelung, auf welches definierende Vorkommen sich ein angewandtes Vorkommen bezieht. Dies regeln die **Gültigkeitsbereichsregeln** und die **Sichtbarkeitsregeln**.

Der **Gültigkeitsbereich** (scope, range of validity) eines definierenden Vorkommens eines Bezeichners x ist der Teil des Programms (oder mehrerer Programme), in dem sich ein angewandtes Vorkommen von x auf dieses definierende Vorkommen beziehen kann.

Die Aufgabe, jedem angewandten Vorkommen eines Bezeichners das gemäß der Gültigkeits- und der Sichtbarkeitsregeln zugehörige definierende Vorkommen (oder die zugehörigen definierenden Vorkommen) zuzuordnen, nennt man die **Identifizierung von Bezeichnern** (identification of identifiers). Wir werden später sehen, daß in Programmiersprachen, die das **Überladen** von Bezeichnern erlauben, sich ein angewandtes Vorkommen eines Bezeichners tatsächlich auf mehrere definierende Vorkommen beziehen kann.

Die Gültigkeits- und die Sichtbarkeitsregeln einer Programmiersprache hängen stark davon ab, welche Art von Schachtelung von Scope-Konstrukten die Sprache erlaubt.

Gültigkeit

Cobol erlaubt keine Schachtelung von Scope-Konstrukten; alle Bezeichner sind überall gültig und sichtbar. Fortran erlaubt nur die Schachtelungstiefe 1, also nicht weiter geschachtelte Blöcke, d.h. Prozedur-/ Funktionsdeklarationen in einem Hauptprogramm. Bezeichner, die in einem Block definiert sind, sind nur innerhalb dieses Blocks sichtbar. Ein im Hauptprogramm deklarierter Bezeichner ist beginnend mit der Deklaration überall sichtbar, außer in Prozedurdeklarationen, die eine neue Deklaration des Bezeichners enthalten.

Algol60, Algol68, PL/I, Pascal, Ada, C und funktionale Programmiersprachen erlauben die unbeschränkt tiefe Schachtelung von Blöcken. Der Gültigkeits- und der Sichtbarkeitsbereich definierender Vorkommen von Bezeichnern wird dann durch zusätzliche Festlegungen geregelt.

Für eine let-Konstrukt **let** $v_1 == e_1; \ldots; v_n == e_n$ **in** e_0 sind die Bezeichner $v_1, \ldots, v_n$ in e_0, dem **Rumpf** des let-Konstrukts gültig. Angewandte Vorkommen der v_i in den Ausdrücken $e_1, \ldots, e_n$ beziehen sich also auf definierende Vorkommen in umfassenden Blöcken. Analoges gilt für das where-Konstrukt.

Der Gültigkeitsbereich der Bezeichner $v_1, \ldots, v_n$ eines letrec-Konstrukts **letrec** $v_1 == e_1; \ldots; v_n == e_n$ **in** e_0 besteht aus den Ausdrücken $e_0, e_1, \ldots, e_n$. Die gleiche Festlegung gilt auch für Algol60- und Algol68-Blöcke. Jeder in einem Block deklarierte Bezeichner ist im ganzen Block gültig.

Diese Regelung macht die 1-Pass-Übersetzbarkeit einer Programmiersprache unmöglich; denn der Übersetzer braucht zur Übersetzung einer Deklaration eventuell Informationen über einen Bezeichner, dessen Deklaration noch nicht bearbeitet wurde. Deshalb gibt es in Pascal und Ada (und C, s.o.) Regeln, die dieses Problem vermeiden. In Ada fängt der Gültigkeitsbereich eines Bezeichners mit dem Ende der Deklaration an und hört mit dem Ende des Blocks auf. In Pascal ist der Gültigkeitsbereich der ganze Block, aber kein angewandtes Vorkommen darf vor dem Ende der Deklaration stehen.

Prolog hat mehrere Klassen von Bezeichnern, die durch ihre syntaktische Position charakterisiert sind. Die Bezeichner aus den verschiedenen Klassen haben die folgenden Gültigkeitsbereichsregeln (da es keine Verdeckung gibt, stimmt Gültigkeit mit Sichtbarkeit überein):

- Prädikate und Funktoren haben globale Gültigkeit; sie sind im ganzen Prolog-Programm und in zugehörigen Anfragen gültig.
- Bezeichner von Klauselvariablen haben Gültigkeit nur in der Klausel, in der sie vorkommen.

Es gibt so etwas wie Deklarationen nur für Prädikate, nämlich die Menge der Alternativen für ein Prädikat. Variablen sind ungetypt und müssen deshalb nicht deklariert werden. Es gibt zwar auch den Sprachgebrauch „definierendes und angewandtes Vorkommmen von Variablen"; dies hat jedoch eine andere Bedeutung; ein definierendes Vorkommen einer Klauselvariable ist eines, das im sequentiellen Ablauf zuerst gebunden wird, ein angewandtes eins, dessen vorher hergestellte Bindung benutzt wird.

Sichtbarkeit

Nicht an jeder Stelle des Gültigkeitsbereichs eines definierenden Vorkommens von x meint ein angewandtes Auftreten von x tatsächlich dieses definierende Vorkommen. Ist das definierende Vorkommen **global** zum aktuellen Block, d.h. nicht in dessen Deklarationsteil, so kann eine lokale Deklaration von x es **verdecken**. Es ist dann nicht **direkt sichtbar**. Es gibt aber mehrere Möglichkeiten, ein nicht direkt sichtbares definierendes Vorkommen eines Bezeichners x innerhalb seines Gültigkeitsbereichs sichtbar zu machen. Die Sichtbarkeitsregeln einer Programmiersprache legen fest, auf welche definierenden Vorkommen eines Bezeichners sich ein angewandtes Vorkommen beziehen kann. Hier sind einige Sichtbarkeitsregeln aus existierenden Programmiersprachen:

- Die **Erweiterung eines Bezeichners** um den Bezeichner eines die Deklaration enthaltenden Konstrukts ermöglicht den Bezug auf ein verdecktes definierendes Vorkommen. In Pascal gibt es die Möglichkeit, einen verdeckten Verbundkomponentennamen durch die Erweiterung um den Verbundnamen sichtbar zu machen. Ada erlaubt die Erweiterung um den Bezeichner der Programmeinheit, in welcher die gewünschte Deklaration steht.

- Einige Direktiven erlauben es, ein verdecktes definierendes Vorkommen eines Bezeichners in einem Teil des Gültigkeitsbereichs, einer **Region**, ohne Bezeichnererweiterung sichtbar zu machen. Diese Direktiven heißen meist auch Anweisungen (statements), sind aber Teil der statischen und nicht der dynamischen Semantik. Die Grenzen der Region sind entweder durch eine eigene Anfang-/Endklammerung bestimmt, wie bei der Pascal-with-Direktive, oder sie sind gleich den Grenzen der sie direkt enthaltenden Programmeinheit. Die use-Direktive in Ada listet Bezeichner von umgebenden Programmeinheiten auf, deren Deklarationen dadurch sichtbar werden. Die Sichtbarkeit dieser Bezeichner erstreckt sich vom Ende der **use**-Direktive bis zum Ende der umfassenden Programmeinheit.

- Kennt eine Sprache (und nicht nur ihre Implementierung) das Konzept der getrennten Übersetzung von Programmeinheiten, so gibt es Direktiven, die Definitionen aus getrennt übersetzten Einheiten sichtbar machen. Jede getrennt übersetzbare Programmeinheit kann einige ihrer Definitionen zur Benutzung anbieten, ein Ada-Paket (ein Modul) etwa die Definitionen aus seinem öffentlichen Teil, und eine Prozedur ihre formalen Parameter. Diesen Bezeichnern ordnen wir einen Gültigkeitsbereich zu, der alle Programme umfaßt, die nach getrennter Übersetzung zusammengebunden werden. Die Ada-with-Direktive macht dann solche von separat übersetzten Einheiten angebotenen und in der with-Liste erwähnten Bezeichner sichtbar.

Zusammenfassend kann man sagen: Der Gültigkeitsbereich eines definierenden Auftretens eines Bezeichners x ist der Teil eines Programms, in dem der Bezeichner benutzt werden kann, um das ihm in der Definition zugeordnete Objekt anzusprechen. Im Gültigkeitsbereich ist die Definition entweder direkt sichtbar, oder sie kann sichtbar gemacht werden.

9.1.2 Überprüfung der Kontextbedingungen

Wir werden nun skizzieren, wie man in Übersetzern die Einhaltung der Kontextbedingungen überprüft. Dazu betrachten wir einen einfachen Fall, eine Programmiersprache mit geschachtelten Scope-Konstrukten ohne Moduln und ohne Überladung.

Die Aufgabe wird in zwei Teilaufgaben zerlegt. Die erste sei durch einen Modul, genannt den Deklarations-Analysator gelöst. Er erledigt das Problem der Identifizierung von Bezeichnern und prüft dabei, ob die Deklariertheitseigenschaften erfüllt sind. Hier gehen die Gültigkeits- und die Sichtbarkeitsregeln der Programmiersprache ein. Die zweite Teilaufgabe überprüft die Typkonsistenz.

Identifizierung von Bezeichnern

Gemäß den Gültigkeits- und Sichtbarkeitsregeln gehört (in unserem einfachen Fall) zu jedem angewandten Vorkommen eines Bezeichners in einem korrekten

Programm genau ein definierendes Vorkommen. Die Identifizierung von Bezeichnern besteht darin, diesen Bezug von angewandten Vorkommen auf definierende Vorkommen herzustellen, bzw. festzustellen, daß kein solcher Bezug oder kein eindeutiger besteht. Das Ergebnis der Identifizierung wird von der Typüberprüfung und der Codeerzeugung benutzt. Deshalb muß es diese Phase überleben. Für die Darstellung der Korrespondenz zwischen angewandten und definierenden Vorkommen gibt es eine Reihe von Möglichkeiten. Traditionell erstellt ein Übersetzer eine sogenannte **Symboltabelle**, in der für jedes definierende Vorkommen eines Bezeichners die zugehörige deklarative Information abgespeichert ist. Diese Symboltabelle ist meist analog zur Blockstruktur des Programms organisiert, so daß man von jedem angewandten Vorkommen (schnell) zu dem korrespondierenden definierenden Vorkommen gelangen kann. Eine solche Symboltabelle ist nicht das Ergebnis der Identifizierung sondern dient nur dazu, diese vorzunehmen. Das Ergebnis der Identifizierung besteht darin, daß bei jedem Knoten für ein angewandtes Vorkommen eines Bezeichners entweder

(1) ein Verweis auf den Knoten für die Deklaration oder

(2) die Adresse des Eintrages für das definierende Vorkommen in der Symboltabelle oder

(3) die deklarative Information zu dem definierenden Vorkommen

abgespeichert ist. Bei den Alternativen (1) und (3) kann die Symboltabelle nach Ende der Identifizierungsphase aufgegeben werden. Der Syntaxbaum, vermehrt um nicht kontextfreie Information, bleibt die einzige Datenstruktur. Die Alternative (1) hat den zusätzlichen Vorteil, daß alle angewandten Vorkommen zu einem definierenden Vorkommen dieses gemeinsam benutzen. Deshalb entscheiden wir uns für die Alternative (1).

Welche Operationen muß die Symboltabelle anbieten? Wenn der Deklarations-Analysator eine Deklaration antrifft, muß er den deklarierten Bezeichner und einen Verweis auf den zugehörigen Deklarationsknoten im Syntaxbaum in die Symboltabelle eintragen. Solch eine Deklaration steht in einem Block. Eine weitere Operation muß das Öffnen von Blöcken vermerken, eine andere das Schließen von Blöcken. Letztere kann die Einträge zu Deklarationen des geschlossenen Blocks aus der Symboltabelle entfernen. Dadurch enthält die Symboltabelle zu jeder Zeit genau die Einträge zu Deklarationen aller zu dieser Zeit geöffneten, aber noch nicht geschlossenen Blöcke. Trifft der Deklarationsanalysator auf ein angewandtes Vorkommen eines Bezeichners, so sucht er die Symboltabelle gemäß den Gültigkeits- und Sichtbarkeitsregeln nach dem Eintrag des zugehörigen definierenden Vorkommens ab. Hat er es gefunden, so kopiert er den dort eingetragenen Verweis auf die Deklarationsstelle zum Knoten für das angewandte Vorkommen.

Damit sind insgesamt die folgenden Operationen auf der Symboltabelle notwendig:

(a)	*create_symb_table*	kreiert eine leere Symboltabelle.
(b)	*enter_block*	vermerkt das Öffnen eines neuen Blocks.
(c)	*exit_block*	setzt die Symboltabelle auf den Stand zurück, den sie vor dem letzten *enter_block* hatte.
(d)	*enter_id(id, decl_ptr)*	fügt einen Eintrag für Bezeichner *id* in die Symboltabelle ein. Dieser enthält den Verweis auf seine Deklarationsstelle, die in *decl_ptr* übergeben wird.
(e)	*search_id(id)*	sucht das definierende Vorkommen zu *id* und gibt den Verweis auf die Deklarationsstelle zurück, wenn er existiert.

Die beiden letzten Operationen bzw. Funktionen arbeiten relativ zum letzten geöffneten Block, dem aktuellen Block.

Bevor die Implementierung der Symboltabelle, d.h. der oben aufgelisteten Prozeduren und Funktionen angegeben wird, wird ihre Benutzung bei der Deklarationsanalyse vorgeführt. Dazu nehmen wir Ada-ähnliche Gültigkeitsregeln an; d.h. ein definierendes Vorkommen eines Bezeichners ist erst ab dem Ende seiner Deklaration gültig.

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


Die Ada-Gültigkeitsregeln drücken sich darin aus, daß in dem *decl*-Fall der case-Anweisung erst alle Deklarationen rekursiv abgearbeitet werden, bevor die lokalen Deklarationen eingetragen werden. Die Modifikation dieses Algorithmus für Algol-ähnliche und Pascal-ähnliche Gültigkeitsregeln bleiben dem Leser überlassen (siehe Übung 1.4).

Überprüfung der Typkonsistenz

Die Überprüfung der Typkonsistenz kann in einem bottom up-Pass über Ausdrucksbäume erfolgen. Für terminale Operanden, die Konstanten sind, steht der Typ schon fest; für Bezeichner besorgt man sich den Typ von seiner Definitionsstelle. Für jeden Operator schlägt man in einer Tabelle nach (siehe Abb. 9.4), ob die Typen der Operanden zu ihm passen und welches der Ergebnistyp ist. Bei der Überladung eingebauter Operatoren wird dabei noch die richtige Operation ausgewählt.

Erlaubt die Programmiersprache Typanpassungen, etwa von integer → real, so wird für jeden Operator und jede Kombination aus Operandentypen, die nicht zu ihm passen, geprüft, ob die Operandentypen durch Typanpassung zu einer für den Operator gültigen Kombination von Operandentypen gemacht werden können.

Implementierung der Symboltabelle

Bei der Implementierung einer Symboltabelle muß man darauf achten, daß die *search_id*-Funktion zu jedem Zeitpunkt von eventuell mehreren möglichen Einträgen für einen Bezeichner den gemäß der Sichtbarkeitsregeln richtigen findet.

Als erste Lösung könnte einem eine lineare Liste aus *enter_block*- und *enter_id*-Einträgen einfallen. Neue Einträge werden hinten angehängt und *exit_block* löscht von hinten alle *enter_id*-Einträge bis einschließlich dem letzten *enter_block*-Eintrag. *search_id* durchsucht die Liste von hinten und wird dabei alle gemäß der Ada-Gültigkeitsregel aktuell gültigen Bezeichner finden. Da diese lineare Liste offensichtlich kellerartig verwaltet wird, kann man sie auch als Keller organisieren.

An dieser Lösung stört der Aufwand für *search_id*, der linear von der Zahl der deklarierten Bezeichner abhängt. Logarithmische Suchzeit in jedem Block wird erreicht, wenn man für jeden Block die Einträge in einem binären Suchbaum verwaltet. Die Suche beginnt dann bei dem Suchbaum des aktuellen Blocks, fährt beim Suchbaum des umfassenden Blocks fort, bis ein definierendes Vorkommen gefunden wird oder festgestellt wird, daß kein solches existiert.

Wenn man davon ausgeht, daß jeder definierte Bezeichner mehrfach angewandt vorkommt, so sollte vor allen Dingen *search_id* sehr effizient, am besten in konstanter Zeit ablaufen. Das läßt sich mit folgender Datenstruktur verwirklichen:

Die Einträge aller aktuell gültigen definierenden Vorkommen eines Bezeichners werden linear verkettet; ein neuer Eintrag wird hinten an diese Kette angefügt. Auf den jeweils letzten eingefügten Eintrag zeigt eine durch den Bezeichner indizierte Komponente eines Feldes. Außerdem sind alle zum gleichen Block gehörenden Einträge verkettet, um die Prozedur *exit_block* zu unterstützen. Auf diese Kette zeigt ein dem Block zugeordneter Listenkopf. Diese Listenköpfe können kellerartig verwaltet werden.

Beispiel 9.1.2
Für das Programm in Abbildung 9.3 und den mit * markierten Punkt ergibt sich die in Abbildung 9.4 dargestellte Symboltabelle. □

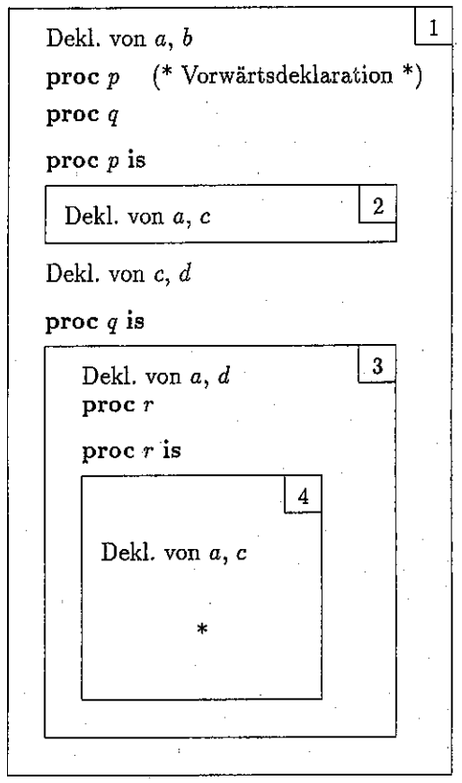

Abb. 9.3: Programmbeispiel

Die Implementierung der Symboltabellenoperationen ist die folgende:

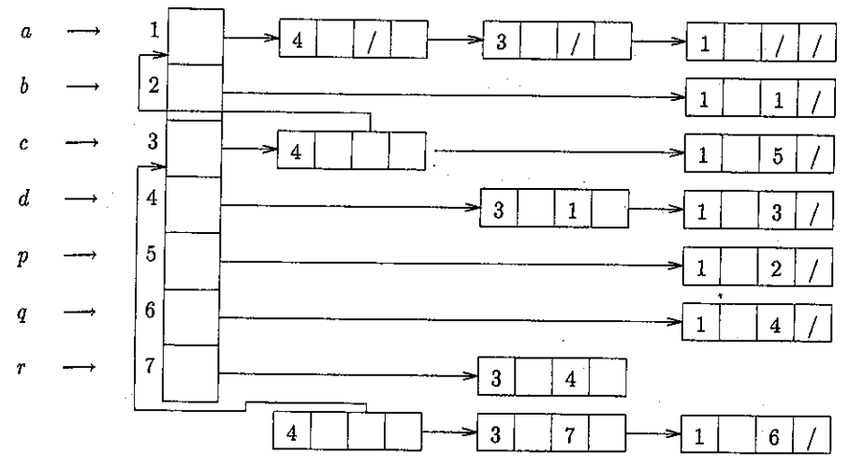

Abb. 9.4: Symboltabelle zum Programm aus Abbildung 9.3. Die Verkettung der Einträge zu einem Block ist nur für Block 4 angegeben, da sonst das Diagramm zu unübersichtlich wird. Sonst sind die „Adressen" der Kästchen, die Zahlen 1 - 7 angegeben. Das nicht ausgefüllte Kästchen in jedem Eintrag enthält jeweils den Verweis auf den Unterbaum für die Deklaration.

```
proc create_symb_table;
  begin
      kreiere leeren Keller von Blockeinträgen
  end;

proc enter_block;
  begin
      kellere neuen Eintrag für den neuen Block
  end;

proc exit_block;
  begin
      foreach Deklarationseintrag des aktuellen Blocks do
          lösche Eintrag
      od;
      entferne Blockeintrag aus dem Keller
  end;
```

```
proc  enter_id ( id: Idno; decl: ↑ node );
    begin
        if  exist. bereits ein Eintrag für id in diesem Block
        then  error("Doppeldeklaration")
        fi;
        kreiere neuen Eintrag mit decl und Nr. des akt. Blocks;
        füge diesen Eintrag hinten an die lineare Liste für id an;
        füge diesen Eintrag hinten an die lineare Liste für diesen Block an
    end;

function  search_id ( id: idno ) ↑ node;
    begin
        if Liste für id ist leer
        then  error("undeklarierter Bezeichner")
        else  return (Wert des decl-Feldes aus erstem Eintrag in Liste id )
        fi
    end
```

9.1.3 Überladung von Bezeichnern

Ein Symbol heißt **überladen**, wenn es an einer Stelle im Programm mehrere Bedeutungen haben kann. Schon die Mathematik kennt überladene Symbole, etwa die arithmetischen Operatoren, die je nach Kontext Operationen auf den ganzen, den reellen oder den komplexen Zahlen oder sogar allgemein in Ringen oder Körpern bedeuten. Entsprechend der mathematischen Tradition haben schon die frühen Programmiersprachen Fortran und Algol60 die arithmetischen Operatoren überladen. Eine Typberechnung, wie sie im vorherigen Abschnitt vorgestellt wurde, dient im Übersetzer dazu, abhängig vom Typ der Operanden und oft auch noch vom verlangten Typ des Ergebnisses die richtige Operation zu einem überladenen Operator auszuwählen.

Programmiersprachen erlauben häufig die Überladung von benutzerdefinierten Symbolen, etwa von Prozedur- und Funktionsnamen. Dann können auch in korrekten Programmen bei einem angewandten Vorkommen eines Bezeichners x mehrere definierende Vorkommen von x sichtbar sein. Eine Redeklaration eines Bezeichners x verbirgt nur dann eine äußere Deklaration von x, wenn beide den gleichen Typ haben. Das Programm ist nur dann korrekt, wenn aufgrund der „Typumgebung" des angewandten Vorkommens genau eines der definierenden Vorkommen ausgewählt werden kann. Die Typumgebung bei Prozedur- oder Funktionsaufrufen besteht dabei in der Kombination der Typen der aktuellen Parameter.

Die Sichtbarkeitsregeln von Ada kombiniert mit den Möglichkeiten für die Überladung von Symbolen erfordern eine kaum überschaubare und verständliche Menge von Konfliktauflösungsregeln für die Fälle, wo auf verschiedene Weise sichtbare oder sichtbar gemachte aber nicht überladene Bezeichner in Konkurrenz stehen.

Beispiel 9.1.3 (Ada-Programm (Istvan Bach))

```
procedure BACH is
    procedure put (x: boolean) is begin null; end;
    procedure put (x: float)   is begin null; end;
    procedure put (x: integer) is begin null; end;
    package x is
       type boolean is (false, true);
       function f return boolean;                              -- (D1)
    end x;
    package body x is
       function f return boolean is begin null; end;
    end x;
    function f return float is begin null; end;                -- (D2)
    use x;
begin
    put (f);                                                   -- (A1)
    A: declare
       f: integer;                                             -- (D3)
    begin
       put (f);                                                -- (A2)
       B: declare
          function f return integer is begin null; end;        -- (D4)
          begin
             put (f);                                          -- (A3)
          end B;
    end A;
end BACH;
```

Das Paket `x` deklariert in seinem öffentlichen Teil zwei neue Bezeichner, nämlich den Typ-Bezeichner `boolean` und den Funktions-Bezeichner `f`. Diese beiden Bezeichner werden durch die use-Anweisung `use x;` (siehe hinter (D2)) ab dem Semikolon (potentiell) sichtbar gemacht. Funktions-Bezeichner sind in Ada überladbar. Da die beiden Deklarationen von `f`, bei (D1) und (D2), verschiedene „Parameterprofile" haben, d.h. in diesem Fall unterschiedliche Ergebnistypen, sind sie beide am Punkt (A1) (potentiell) sichtbar.

Die Deklaration `f: integer` in der Programmeinheit `A` (siehe (D3)) verdeckt die äußere Deklaration (D2) von `f`, da Variablen-Bezeichner in Ada nicht überladbar sind. Aus diesem Grunde ist auch die Deklaration (D1) nicht sichtbar. Die Deklaration (D4) von `f` in der Programmeinheit `B` verdeckt wiederum die Deklaration (D3), und da diese die Deklaration (D2) verdeckt, transitiv auch diese. Die durch die use-Anweisung (potentiell) sichtbar gemachte Deklaration (D1) wird allerdings nicht verdeckt, sondern ist nach wie vor potentiell sichtbar. Im Kontext `put(f)` (siehe (A3)) kann sich `f` nur auf die Deklaration (D4) beziehen, da die erste Deklaration von `put` einen anderen Typ, `boolean`, benutzt als der Ergebnistyp von `f` in (D1). □

Die Auswahl des richtigen definierenden Vorkommens eines überladenen Symbols nennt man die **Auflösung der Überladung** (overload resolution). Die Auflösung von Überladungen findet nach der Identifizierung von Bezeichnern innerhalb bestimmter Konstrukte der Sprache statt, d.h. beschränkt auf Ausdrücke, (zusammengesetzte) Bezeichner etc.

Der Auflösungsalgorithmus läuft auf der Darstellung des Ada-Programms als abstraktem Syntaxbaum ab. Konzeptionell benutzt er dabei vier Läufe über einen Ausdrucksbaum. Allerdings läßt sich der erste mit dem zweiten und der dritte mit dem vierten verschmelzen.

Um den Algorithmus zu formulieren, führen wir die folgende Notation ein: An jedem Knoten k des abstrakten Syntaxbaums erhalten wir über

$\#descs(k)$	die Zahl der Kindknoten von k,
$symb(k)$	das Symbol mit dem k markiert ist,
$vis(k)$	Menge der an k sichtbaren Definitionen von $symb(k)$
$ops(k)$	die Menge der augenblicklichen Kandidaten für das überladene Symbol $symb(k)$ und
$k.i$	wie üblich, das i-te Kind von k.

Für jedes definierende Vorkommen eines überladenen Symbols op mit Typ $t_1 \times \cdots \times t_m \to t$ sei

$$\begin{aligned} rank(op) &= m \\ res\_typ(op) &= t \\ par\_typ(op,i) &= t_i \quad (1 \le i \le m). \end{aligned}$$

Die beiden letzteren erweitern wir auf Mengen von Operatoren. Für jeden Ausdruck, in welchem die Überladung von Operatoren aufgelöst werden soll, wird aus seinem Kontext ein Typ, der sogenannte a-priori-Typ, berechnet.

```
proc  resolve_overloading (root: node, a_priori_type: type);

func pot_res_types (k: node): set of type;
                (* potentielle Typen des Resultats *)
    return {res_typ(op) | op ∈ ops(k)}

func act_par_types (k: node, i: integer): set of type;
    return {par_typ(op,i) | op ∈ ops(k)}

proc  init_ops
begin
    foreach  k
        ops(k) := {op | op ∈ vis(k) und  rank(op) = #descs(k)}
    od;
    ops(root) := {op ∈ ops(k) |  res_typ(op) = a_priori_typ}
end;

proc  bottom_up_elim (k: node);
begin
    for  i := 1 to  #descs(k) do
```



```
        bottom_up_elim (k.i);
        ops(k) := ops(k) − {op ∈ ops(k) |  par_typ(op,i) ∉ pot_res_types(k.i)}
                (* entferne die Operatoren, deren i-ter Parametertyp zu keinem
                der möglichen Resultattypen des i-ten Operanden paßt *)
    od;
end;

proc  top_down_elim (k: node);
begin
    for  i := 1 to  #descs(k) do
        ops(k.i) := ops(k.i) − {op ∈ ops(k.i) | res_typ(op) ∉ act_par_types(k,i)};
                (* entferne die Operatoren, deren Resultattyp nicht zu
                irgendeinem Typ des zugehörigen Parameters paßt *)
        top_down_elim(k.i)
    od;
end;

begin
    init_ops;
    bottom_up_elim(root);
    top_down_elim(root);
    prüfe, ob jetzt alle ops-Mengen einelementig sind; sonst Fehlermeldung
end
```

Es sieht so aus, als ob die bottom up-Elimination und die top down-Elimination das Gleiche täten. Das ist auch fast richtig. Abbildung 9.5 zeigt eine Kombination von op_1- und op_2- markierten Knoten. Mit jedem der Knoten ist eine Menge von möglichen Definitionen des Operators assoziiert. Die bottom up-Elimination löscht eventuell Kandidaten aus der Definitionsmenge von op_1, die top down-Elimination aus der von op_2.

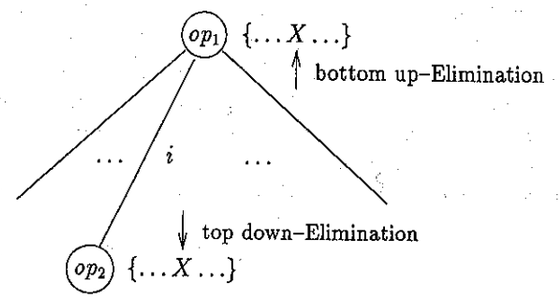

Abb. 9.5:

9.1.4 Polymorphismus

Überladung von Operatoren, wie im letzten Abschnitt informell eingeführt, erlaubt die Sichtbarkeit verschiedener Definitionen bei einer Anwendung. Eine der Definitionen muß aufgrund der Operanden- und Ergebnistypkombination ausgewählt werden. Man bezeichnet die Überladung von Operatoren auch als **ad-hoc-Polymorphismus**. Im Gegensatz dazu erlaubt es der **parametrische Polymorphismus**, der in funktionalen Programmiersprachen und auch in Form der generischen Pakete in Ada vorkommt, **eine** Definition einer Funktion zu geben, die für eine Menge von Kombinationen von Operanden- und Ergebnistypen im wesentlichen dasselbe tut. Man zahlt für diese Flexibilität nicht mit schlechterer Fehlerentdeckung; denn Teil jedes Übersetzers für eine Programmiersprache mit polymorphen Typen ist ein Typinferenzalgorithmus, der für jede Anwendung einer polymorph getypten Funktion den richtigen Typ ausrechnet und feststellt, ob all diese so berechneten Typen ein gemeinsames Typschema haben, welches dann der Typ der Funktion ist.

Der Typinferenzmodul ordnet jeder in einem typkorrekten Programm definierten Funktion ihren allgemeinsten Typ zu, d.h. die am wenigsten festgelegten Operanden- und Ergebnistypen, unter denen bei keiner dynamischen Ausführung des Programms ein Typfehler bei einer Anwendung der Funktion auftreten kann. Dabei können statisch verschiedenen Anwendungen der Funktion durchaus verschiedene Instanzen dieses allgemeinsten Typs zugeordnet werden.

Wir benutzen als Beispiel die Sprache LaMa aus Kapitel 3, Abschnitt 3.2. Wir wiederholen die Darstellung der (abstrakten) Syntax von LaMa-Programmen in Tabelle 9.3. Es wurde ein let-Konstrukt hinzugenommen.

Typterme (kürzer **Typen**) werden aufgebaut aus Typvariablen, $\alpha, \beta, \gamma, \ldots$, die für beliebige Typen[1] stehen, und aus Operatoren. Die eingebauten Typen *int* und *bool* werden durch die nullstelligen Operatoren **int** und **bool** dargestellt. **list** ist ein einstelliger Operator. $\rightarrow$, der Konstruktor für Funktionstypen, und $\times$, der Konstruktor für Paartypen, sind zweistellige Operatoren. Die allgemeinste Form von Listen-, Paar- und Funktionstypen ist **list** α, $\alpha \times \beta$ bzw. $\alpha \rightarrow \beta$. Durch Einsetzen von Typen für die Typvariablen, genannt Instantiierung, gewinnt man speziellere Typen, etwa **list int** bzw. **int** $\rightarrow$ **bool**, **list** $\alpha \rightarrow$ **int** und $(\gamma \rightarrow \delta) \rightarrow \beta$. Typen, die Typvariablen enthalten, heißen **polymorph**, solche ohne Typvariablen **monomorph**. Die polymorphe Identitätsfunktion $\lambda x.x$ hat den Typ $\alpha \rightarrow \alpha$. Von ihr gibt es die monomorphen Instantiierungen **int** $\rightarrow$ **int**, **bool** $\rightarrow$ **bool**, **list int** $\rightarrow$ **list int** usw.

Die Zuordnung von eingebauten LaMa-Operatoren zu Typen ist die folgende:

true, false : bool
$0, 1, \ldots :$ **int**
succ, pred, neg : int $\rightarrow$ **int**
pair: $\alpha \rightarrow \beta \rightarrow (\alpha \times \beta)$
fst: $(\alpha \times \beta) \rightarrow \alpha$
snd: $(\alpha \times \beta) \rightarrow \beta$

[1] Beachten Sie, daß wir den Begriff Typ jetzt überladen haben: Typen sind sowohl die eingebauten und die benutzerdefinierbaren Typen der Programmiersprache, also semantische Konzepte, als auch deren Darstellung durch Typterme.

plus, sub, mul, div : int $\rightarrow$ **int** $\rightarrow$ **int**
cons : $\alpha \rightarrow$ **list** $\alpha \rightarrow$ **list** α
head : **list** $\alpha \rightarrow \alpha$
tail : **list** $\alpha \rightarrow$ **list** α
null : **list** $\alpha \rightarrow$ **bool**

Tabelle 9.3: Die Syntax von LaMa.

Elementname	Bereich	
b	B	Menge von Basiswerten, z.B. boolesche Werte, integer, character,... Die leere Liste, []
op_{bin}	Op_{bin}	Menge von binären Operatoren, z.B. $+, -, =, \neq$, **and, or, cons, pair**,...
op_{un}	Op_{un}	Menge von unären Operatoren, z.B. $-$, **not, head, tail, fst, snd**,...
v	V	Menge von Variablen
e	E	Menge von Ausdrücken

$e = b \mid v \mid (op_{un}\ e\ ) \mid (e_1\ op_{bin}\ e_2)$
$\mid$ (**if** e_1 **then** e_2 **else** e_3)
$\mid (e_1 e_2)$ Funktionsanwendung
$\mid (\lambda v.e)$ funktionale Abstraktion
$\mid$ (**letrec** $v_1 == e_1$; simultan rekursive Definitionen
$v_2 == e_2$;
$\vdots$
$v_n == e_n$
in e_0)
$\mid$ (**let** $v_1 == e_1$; nicht rekursive Definition
$v_2 == e_2$;
$\vdots$
$v_n == e_n$
in e_0)

Diese Zuordnung beschreibt eine initiale **Typumgebung**, mit der die Typinferenz beginnt. Den zusammengesetzten LaMa-Konstrukten werden **Typkombinationsregeln** zugeordnet, welche Bedingungen auf den Typen der Konstituenten beschreiben. Die Anwendung der Regeln führt neue Typvariablen ein.

Der bedingte Ausdruck **if** e **then** e_1 **else** e_2 **fi** hat den Typ α und liefert die Typbedingungen e : **bool** und $e_1 : \alpha$ und $e_2 : \alpha$ mit einer neu eingeführten Typvariable α. Die für e, e_1, e_2 ausgerechneten Typen t, t_1 und t_2 müssen also diese Bedingungen erfüllen. Die Anpassung zwischen diesen errechneten Typen und den obigen Typen geschieht durch Unifikation. Dabei können sowohl α als auch Typvariable in den t, t_1 und t_2 gebunden werden. Natürlich müssen alle Vorkommen von α an den gleichen Typ gebunden werden.

Die **Funktionsanwendung** $e_1\, e_2$ verlangt, daß e_1 einen Typ $\alpha \to \beta$ und e_2 den Typ α hat, und ergibt für die Anwendung den Typ β, wieder mit neuen Typnamen α und β.

Die **Abstraktion** $\lambda v.e$ hat den Typ $\alpha \to \beta$, wobei α eine neue Typvariable für den Typ von v und β der Typ ist, der sich für e ergibt, wenn **alle** freien Vorkommen von v in e den Typ α zugeordnet bekommen. Es sieht so aus, als ob die Festlegung, daß alle freien Vorkommen von v in e denselben Typ zugeordnet bekommen, eine unnötige Einschränkung der Flexibilität des Typsystems wäre; denn schließlich sollte es doch möglich sein, verschiedenen Vorkommen eines polymorph getypten Bezeichners verschiedene Typen zuzuordnen. Betrachten wir folgendes Beispiel: $g = \lambda f.$**pair** $(f\ 5)$ $(f$ **true**$)$. Der Teilausdruck $f\ 5$ belegt f mit dem Typ **int** $\to \alpha$, der Ausdruck f **true** mit **bool** $\to \beta$; Die Typen **int** $\to \alpha$ und **bool** $\to \beta$ sind nicht miteinander unifizierbar. Deshalb kann dem obigen Ausdruck kein Typ zugeordnet werden. Natürlich gibt es Typzuordnungen, die in manchen Fällen korrekt funktionieren, etwa die Zuordnung $g : (\alpha \to$ **int**$) \to$ **int** für die Anwendung $g\ (\lambda x.0)$. Die Anwendung $g\ (\lambda x.0)$ würde nicht zu einem Typfehler zur Laufzeit führen. Das Ergebnis wäre **pair** $(\lambda x.0\ 5)$ $(\lambda x.0$ **true**$)$ = **pair** 0 0. Dies besagt aber nur, daß eine Anwendung auf **diese** Funktion, $\lambda x.0$, dynamisch typkorrekt wäre; verlangt ist aber, daß keine Anwendung von g zu Typfehlern führt. Schon die Anwendung auf die Identitätsfunktion, $g\ (\lambda x.x)$, würde zu einem Typfehler führen, da zur Laufzeit **eine** monomorphe Version der Identitätsfunktion, also etwa der Typen **int** $\to$ **int** oder **bool** $\to$ **bool** übergeben würde. Jede monomorphe Version wird aber zu einem Typfehler führen. Es kommt also bei der Typinferenz nicht darauf an, ob einem Term in speziellen günstigen Kontexten ein Typ zugeordnet werden kann, sondern die Typzuordnung muß unter allen Umständen eine korrekte Typung ergeben.

Deshalb verlangen wir, daß alle von einer λ-Abstraktion gebundenen Vorkommen eines Bezeichners den gleichen Typ zugeordnet bekommen. Die Typvariablen in dem Typ eines durch eine λ-Abstraktion gebundenen Bezeichners werden in dem Gültigkeitsbereich dieser Bindung als **nicht generisch** bezeichnet.

In **let-Ausdrücken let** $v == e_1$ **in** e_2 kann man wieder liberaler sein und verschiedenen Vorkommen von v in e_2 verschiedene Typen zugestehen, da man mehr Informationen als im Fall der λ-Abstraktion hat. Die Bindung von v, nämlich an e_1, ist ja bereits sichtbar. Deshalb läßt sich in

$$\textbf{let } f == \lambda x.x \textbf{ in pair } (f\ 5)\ (f \textbf{ true})$$

f der Typ $\alpha \to \alpha$ zuordnen. Die Typvariablen in den Typen von **let**-gebundenen Bezeichnern, die nicht in den Typen von in der Umgebung des **let** λ-gebundenen

Bezeichnern auftreten, nennen wir **generisch**. Verschiedene Vorkommen des **let**-gebundenen Bezeichners bekommen verschiedene „Versionen" des Typs des Bezeichners zugeordnet, zwei Versionen unterscheiden sich nur in der konsistenten Umbenennung aller generischen Bezeichner in dem Typ. Die generischen Bezeichner liegen fest und müssen deshalb in allen Versionen gleich sein.

Der Typ eines **let**-Ausdrucks

$$\textbf{let } v_1 == e_1; \ldots; v_n == e_n \textbf{ in } e$$

wird also folgendermaßen berechnet: Erst werden alle Gleichungen typgeprüft, wobei als Ergebnis n Paare $v_i : t_i$ anfallen, (t_i Typ von e_i). Für alle Vorkommen von v_i in e setzt man dann verschiedene Versionen von t_i $(1 \leq i \leq n)$ ein und berechnet anschließend den Typ von e.

In einem **letrec-Ausdruck**

$$\textbf{letrec } v_1 == e_1; \ldots; v_n == e_n \textbf{ in } e$$

bekommen die Vorkommen der v_i in den e_i nicht generische Typvariablen und die in e generische zugeordnet.

Die Typüberprüfung kreiert erst eine Typumgebung $\{v_1 : \alpha_1, \ldots, v_n : \alpha_n\}$ mit nichtgenerischen Typvariablen $\alpha_1, \ldots, \alpha_n$. Dann berechnet sie in dieser Typumgebung die Typen t_i der e_i. Darauf werden diese Typen t_i mit den α_i bzw. ihren Instanzen unifiziert.

Beispiel 9.1.4 (Typberechnung für einen LaMa-Ausdruck)

```
letrec  append == λl1 l2.
                    if  null l1
                    then  l2
                    else
                       let   x == head l1
                             y == tail l1
                       in    cons x ( append y l2)
in  ...
```

Typzuordnungen $(a : t)$ ergeben sich für Teilterme des Programms, und Typgleichungen $(\alpha = t)$ für eingeführte Typvariablen α.

(1)	**null** : **list** $\alpha \to$ **bool**	die Typzuordnungen der eingebauten
(2)	**cons** : $\beta \to$ **list** $\beta \to$ **list** β	in der Definition von *append*
(3)	**head**: **list** $\gamma \to \gamma$	auftretenden Operatoren
(4)	**tail** : **list** $\delta \to$ **list** δ	
(5)	*append* : ε	neue Typvariablen für die
(6)	l_1 : ζ	letrec- bzw. λ-gebundenen Namen.
(7)	l_2 : η	

(8)	(**head** l_1) γ	Unifikation zwischen **list** γ als Argumenttyp
	ζ = **list** γ	von **head** und ζ als Typ von l_1.
(9)	(**tail** l_1) : **list** γ	Unifikation zwischen **list** δ und dem neuen
	$\delta = \gamma$	Typ, **list** γ, von l_1.
(10)	$x : \gamma$	linke Seiten erben den errechneten Typ
(11)	y : **list** γ	der rechten Seiten.
(12)	*append*: $\vartheta \to \iota$	Funktionstyp, da links in einer Anwendung.
	$\varepsilon = \vartheta \to \iota$	
	ϑ = **list** γ	Unifikation mit Typ von y.
(13)	(*append* y) : ι	Typ der Anwendung
(14)	(*append* y) : $\kappa \to \mu$	Funktionstyp, da angewendet auf l_2.
	$\iota = \kappa \to \mu$	
	$\kappa = \eta$	Unifikation mit Typ von l_2.
(15)	(*append* y l_2) : μ	
(16)	**cons**: $\gamma \to$ **list** $\gamma \to$ **list** γ	Unifikation von β mit γ.
	$\beta = \gamma$	
(17)	(**cons** x): **list** $\gamma \to$ **list** γ	
	μ = **list** γ	Unifikation mit Typ von (*append* y l_2)
	$\iota = \eta \to$ **list** γ	nach (14)
	ε = **list** $\gamma \to \eta \to$ **list** γ	
(18)	**cons** x (*append* y l_2) : **list** γ	
	η = **list** γ	**then**- und **else**-Teil
	ε = **list** $\gamma \to$ **list** $\gamma \to$ **list** γ	müssen gleichen Typ haben.

Der polymorphe Typ von *append* ergibt sich also zu:

$$\textbf{list}\ \gamma \to \textbf{list}\ \gamma \to \textbf{list}\ \gamma$$

□

9.2 Attributgrammatiken

Das in den meisten Systemen zur Generierung von Übersetzern verwendete Beschreibungssmittel für die statische semantische Analyse sind die Attributgrammatiken (attribute grammars). Sie assoziieren Attribute als Träger statischer semantischer Information mit den Symbolen einer kontextfreien Grammatik, der sogenannten **zugrundeliegenden Grammatik**. Zusätzlich geben sie an, wie die **funktionalen Abhängigkeiten** zwischen den Werten von **Attributvorkommen** in den Produktionen der Grammatik aussehen. Solch eine funktionale Abhängigkeit kann als eine Berechnungsvorschrift aufgefaßt werden, die festlegt,

wie sich der Wert eines Attributvorkommens aus den Werten anderer Attributvorkommen der gleichen Produktion errechnet.

Geeignete Bedingungen für die funktionalen Abhängigkeiten stellen sicher, daß alle **Attributexemplare** in jedem Syntaxbaum zu einem syntaktisch und statisch semantisch korrekten Programm **ausgewertet** werden können, d.h. einen Wert aus ihrem **Attributwertebereich** zugeordnet bekommen können.

Definition 9.2.1 (Attributgrammatik)
Sei $G = (V_N, V_T, P, S)$ eine kontextfreie Grammatik. Wie schon früher schreiben wir die p–te Produktion in P als $p : X_0 \to X_1 \ldots X_{n_p}$, $X_i \in V_N \cup V_T$, $1 \le i \le n_p$, $X_0 \in V_N$. Eine **Attributgrammatik AG** über G besteht aus

- zwei disjunkten Mengen *Inh* und *Syn* von **ererbten** (inherited) bzw. **abgeleiteten** (synthesized) Attributen,
- einer Zuordnung von zwei Mengen $Inh(X) \subseteq Inh$ und $Syn(X) \subseteq Syn$ zu jedem Symbol aus $V_N \cup V_T$; wir bezeichnen mit $Attr(X) = Inh(X) \cup Syn(X)$ die Menge aller Attribute von X; ist $a \in Attr(X_i)$, so hat a ein **Vorkommen** in Produktion p beim Vorkommen von X_i, geschrieben a_i. $V(p)$ sei die Menge aller Attributvorkommen in Produktion p.
- der Festlegung eines **Wertebereichs** D_a für jedes Attribut a;
- der Angabe einer **semantischen Regel**

$$a_i = f_{p,a,i}\,(\,b^1_{j_1}, \ldots, b^k_{j_k}\,) \qquad (0 \le j_l \le n_p)\ (1 \le l \le k)$$

für jedes Attribut $a \in Inh(X_i)$ für $1 \le i \le n_p$ und jedes $a \in Syn(X_0)$ in jeder Produktion p, wobei $b^l_{j_l} \in Attr(X_{j_l})$ $(0 \le j_l \le n_p)$ $(1 \le l \le k)$. $f_{p,a,i}$ ist also eine Funktion von $D_{b^1} \times \ldots \times D_{b^k}$ in D_a.

□

Definition 9.2.2 (def., angew. Vorkommen, Normalform)
AG sei eine Attributgrammatik, p sei eine Produktion. Die Attributvorkommen a_i mit $a \in Inh(X_i)$ und $1 \le i \le n_p$ und mit $a \in Syn(X_0)$ heißen **definierende** Vorkommen. Alle anderen heißen **angewandte** Vorkommen. *AG* ist in **Normalform**, wenn alle Argumente semantischer Regeln angewandte Vorkommen sind. □

Wir haben auch bei den Terminalsymbolen der Grammatik abgeleitete Attribute zugelassen. Andererseits sind die Vorkommen dieser Attribute angewandt. Somit gibt es für sie keine semantischen Regeln. Wie erhalten sie also ihre Werte? In einer formalen Definition der Attributgrammatiken benutzt man dazu sogenannte **externe Regeln**, die die Werte von abgeleiteten Attributexemplaren bei Terminalsymbolen „aus dem Nichts" berechnen. In der Praxis, d.h. in einem Übersetzer, läuft die semantische Analyse nach der lexikalischen und der syntaktischen Analyse ab. Typische abgeleitete Attribute bei Terminalen sind der Wert von Konstanten, die Externdarstellung oder eine eindeutige Identifizierung