Kapitel 5

Kontextanalyse

In der Kontextanalyse werden alle Regeln der Quellsprache überprüft, die weder lexikalisch (im *Scanner*) noch kontextfrei (im *Parser*) überprüft werden können. Die *Kontextbedingungen* einer Sprache werden in einigen Lehrbüchern auch *statisch-semantische Bedingungen* genannt. Das kommt daher, dass vielfach die formale Semantik von Programmiersprachen für alle abstrakten Syntaxbäume einer Sprache definiert wird, und davon können einige Eigenschaften statisch bestimmt werden, also zur „Übersetzungszeit", bevor das Program ausgeführt wird.

Die Kontextbedingungen einer Sprache können mit *Attributgrammatiken* beschrieben werden; daraus lassen sich Programme ableiten, die diese Bedingungen durch *Baumtransduktion* effizient berechnen.

Im Folgenden werden wir typische Aufgaben der Kontextanalyse benennen und die dazu benutzten Begriffe einführen(Abschnitt 5.1) und Attributgrammatiken informell einführen (Abschnitt 5.2). Dann werden wir zwei typische Aufgaben mit Attributgrammatiken beschreiben: Deklarationsanalyse (Abschnitt 5.3) und Typanalyse (Abschnitt 5.5). Zwischendurch werden wir die effiziente Implementierung von Dekalrationstabellen behandeln (Abschnitt 5.4) und schließlich die Umsetzung der Attributgrammatiken in Baumtransduktoren beschreiben, und zwar in funktionalen und objektorientierten Sprachen (Abschnitt 5.6).

5.1 Aufgabe

5.1.1 Beispiel: Kontextbedingungen

Betrachten wir die folgende Zuweisung in einer imperativen Sprache wie Pascal:

$$\ldots a[i,j] := f(1) + 2 : \ldots$$

Diese Programmstelle ist nur dann *wohlgeformt*, wenn folgende Kontextbedingungen erfüllt sind:

1. Die Bezeichner a, i, j und f müssen an dieser Stelle eine gültige Vereinbarung (*Deklaration*) haben.
2. Dabei muss a ein zweidimensionales Feld bezeichnen und eine Variable sein, weil ihr etwas zugewiesen wird. Nehmen wir an, der *Typ* von a sei $I \times J \to E$, wobei I und J die *Indextypen* und E der Elementtyp des Feldes ist.

3. Die Namen i und j müssen Variablen oder Konstanten bezeichnen; ihre Typen müssen mit den Indextypen I bzw. J *verträglich* sein.
4. Der Name f muss eine Funktionsprozedur mit einem Parameter bezeichnen. Ihr Typ hat also die Form $P \to R$, wobei P der Parametertyp und R der Resultattyp ist.
5. Der Parametertyp P muss mit dem Typ des Literals 1 (vermutlich Integer) verträglich sein.
6. Der Resultattyp R von f muss mit dem Operandentyp von + verträglich sein; der ist vermutlich ebenfalls Integer, da der andere Operand von + ebenfalls von diesem Typ ist.
7. Schließlich muss der Elementtyp E von a mit dem Ergebnistyp von + verträglich sein; der ist vermutlich ebenfalls Integer.

Das Beispiel zeigt die zwei typischen Aufgaben der Kontextanalyse:

- Bei der *Deklarationsanalyse* wird überprüft, ob es in einem Programm zu jeder *Benutzung* eines Namens auch eine *Deklaration* gibt.
- Bei der *Typanalyse* wird überprüft, ob die Typregeln der Sprache eingehalten werden, d.h. ob die aktuellen Parameter von Operationen und Prozeduren mit ihren formalen Parametertypen verträglich sind.

Höhere Programmiersprachen sind *blockorientiert*. Blöcke sind in einnder geschachtelte Programmteile (wie Pakete, Moduln, Klassen, Prozeduren), in denen Vereinbarungen getroffen werden können. In einem Block können auch Namen neu vereinbart werden, die in einem äußeren Block bereits vereinbart waren. Dann verdeckt die lokale die äußere Vereinbarung des Namens.

Höhere Programmiersprachen sind *statisch getypt*. Für jede Größe des Programms (Konstante, Variable, Prozedur usw.) wird bei der Vereinbarung ein Typ festgeleg, so dass die Größen werden während der Programmausführung immer Werte des vereinbarten Typs haben.[1] In den Ausdrücken eines Programms muss dann überprüft werden, ob die Typregeln der Sprache eingehalten werden.

Die Kontextanalyse überführt einen abstrakten Synatxbaum in einen *attributuierten Syntaxbaum* oder *Kontextgraphen*, der neben der syntaktischen Struktur noch Querverweise enthält, z. B. von den Benutzungen eines Bezeichners zu dessen Deklaration, und von Ausdrücken zu deren Typ (-definition). Bei rekursiven Typen und Prozeduren kann der Kontextgraph eines Programms durchaus zylisch sein.

5.2 Attributgrammatiken

Attributgrammatiken wurden von Donald Knuth zur Definition der *Semantk kontestfreier Sprachen* eingeführt [1].

...

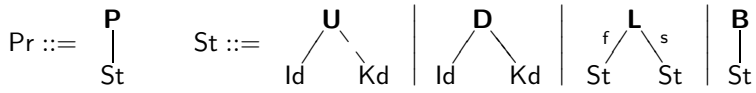
Bemerkung: *Hier fehlen einige formale Definitionen, die ich jetzt weglasse, weil die Beispiele auch ohne sie verstanden werden können. (Hoffe ich jedenfalls.)*

[1] In manchen Sprachen, z. B. den objektorientierten, ist der Sachverhalt etwas komplizierter: dort kann der "dynamische Typ" eines Wertes ein beliebiger Untertyp dieses "statischen Typs" sein.

5.3 Deklarationsanalyse

Für die Deklarationsanalyse betrachten wir eine sehr stark vereinfachte Syntax von Programmiersprachen, in der Anweisungen (*statements*, St) entweder Benutzungen und Vereinbarungen von Bezeichnern, Blöcke oder Sequenzen von Anweisungen sind.

Die abstrakte Syntax besteht also aus den Bäumen



Hier steht Id für die Schlüssel von Bezeichnern, und Kd für die hier nicht näher betrachtete Art von Vereinbarung (eng. *kind*). Das zweite Kind des Knotens **U** ist im Syntaxbaum nicht vorhanden und soll danach die für den Bezeichner gültige Deklaration beinhalten; dieses Kind wird während der Kontextanalyse berechnet und macht den Syntaxbaum zum Kontextgraph.

In einer funktionalen Sprache wie Haskell können solche Bäume ganz leicht mit algebraischen Datentypen beschrieben werden:

```
data Pr = P St
data St = U Id | D Id Kd | L St St | B St
```

In einer objektorientierte Sprache würde man eine Klasse St definieren (ohne Attribute), mit Unterklassen **U**, **D**, **L** und **B**, die Attribute für jedes Kind des entsprechenden Knotens enthalten.

Als zweites legen wir die *semantische Basis* für die Berechnungen auf dem Syntaxbaum fest. Das ist ein abstrakter Datentyp, der Typen und Funktionen zur Verfügung stellt, wie sie für die Deklarationsanalyse benötigt werden. Der Datentyp definiert den Typ `TAB` mit folgenden Operationen:

- initial liefert den Startzustand der Tabelle, mit allen vordefinierten Namen.
- enter fügt eine Deklaration (Kd) für einen Bezeichner zur Tabelle hinzu.
- def sucht die zu einem Bezeichner gehörende Deklation in der Tabelle.
- nest eröffnet einen neuen Block
- unnest verlässt einen Block und löscht alle seine lokalen Deklarationen.

In Haskell würde diese Schnittstelle so aussehen:

```
module DeclarationTable
where type TAB = [[(Id,Kd)]]
      type Id = String
      data Kd = Unbound | Typ | Const | Var | Proc deriving (Eq, Text)
      initial :: TAB
      nest, unnest :: TAB -> TAB
      enter :: TAB -> (Id,Kd) -> TAB
      def :: TAB -> Id -> Kd
```

Mit der effizienten Implementierung dieses Datentyps werden wir uns in Abschnitt 5.4 beschäftigen.

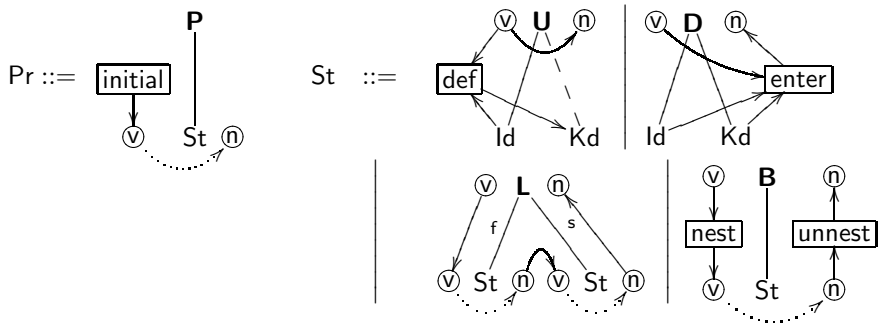
Nun können wir definieren, wie die Deklarationstabelle währen der Traversierung des Baumes aufgebaut und benutzt wird, um das zweite Kind von **U** zu bestimmen. Zunächst behandeln wir den einfacheren Fall, dass in der Sprache *lineare Sichtbarkeit* von Vereinbarungen gilt. D. h., in einem Programm werden immer nur die links von einer Benutzung stehenden Vereinbarungen berücksichtigt. Danach entwickeln wir daraus eine Attributgrammatik für *totale Sichtbarkeit*, bei der alle Vereinbarungen eines Blocks gleichzeitig von Anfang an gültig sind

5.3.1 Deklarationsanalyse für lineare Sichtbarkeit

Wir ordnen jedem Knoten im Syntaxbaum *Attribute* zu, die die Deklarationstabelle beinhalten (genauer gesagt: einen Zustand der Tabelle). Jeder St-Knoten bekommt zwei Attribute vom Typ TAB: Das erste (namens v) enthält alle Vereinbarungen *vor* dem Knoten, das zweite (namens n) alle *nach* dem Knoten gültige Vereinbarungen.

Die Auswertungsregeln für die Attribute in den verschiedenen Knoten des Baumes ergeben sich dann recht einfach:

- Für den Programmknoten **P** wird das Attribut v auf initial gesetzt. Das Attribut n wird nicht gebraucht.
- Beim **U**-Knoten wird das Attribut v und der Bezeichner benutzt, um mit der Funktion def das zweite Kind zu berechnen. Dem Attribut n wird das unveränderte Attribut v zugewiesen.
- Beim **D**-Knoten wird dem Attribut n das mit enter um die gefundene Deklaration erweiterte Attribut v zugewiesen.
- Im **L**-Knoten werden die Attribute von links nach rechts durchgereicht.
- Im **B**-Knoten wird vor der Analyse des Kindknotens mit nest ein neuer Block eröffnet, und nach Durchlauf des Kindknotens mit unnest wieder geschlossen.



In den Regeln bedeuten gestrichelte Pfeile indirekte Abhängigkeiten von Attributen. In diesem Fall geben sie an, welche Abhängigkeiten bei der Auswertung von Unterbäumen bestehen (nämlich jeweils von v nach n).

Die Attributgrammatik kann leicht in rekursive Prozduren idfy' und idfy und umgesetzt werden, die die spezifizierten Attributberechnungen durchführen.

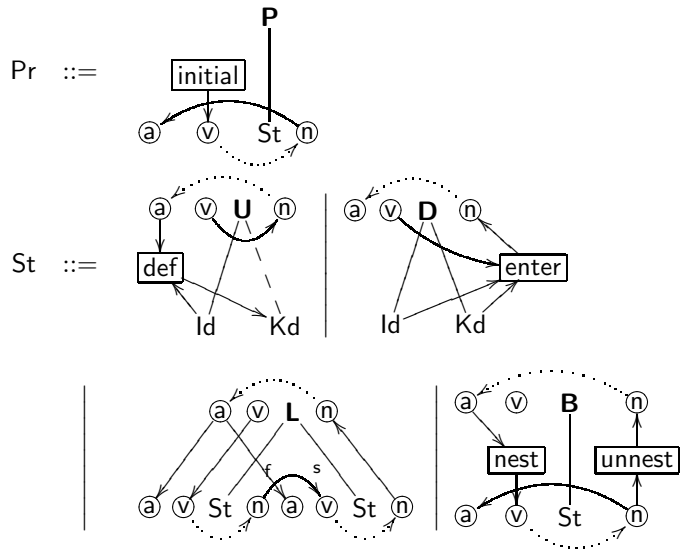
```
idfy' :: Pr -> Pr
idfy' (P s) = P s' where (s',_) = idfy (initial) s
```

```
idfy:: TAB -> St -> (St,TAB)
idfy v (U x _  ) = (U x k',n) where k' = def v x; n = v
idfy v (B s    ) = (B s',unnest n) where (s',n) = idfy (nest v) s
idfy v (D x k  ) = (D x k, n) where n = enter v (x,k)
idfy v (L s1 s2) = (L s1' s2',n') where (s1',n)  = idfy v s1
                                       (s2',n') = idfy n s2
```

5.3.2 Deklarationsanalyse für totale Sichtbarkeit

Wenn zur Identifizierung eines Bezeichners alle Deklarationen neines Blockes berücksichtigt werden müssen, auch wenn sie weiter rechts stehen, müssen die Attribute und Regeln für St-Knoten ergänzt werden:

- Es muss ein Attribut (namens a) hinzugefügt werden, das *alle* Deklarationen eines Blockes enthält.
- Dieses Attribut wird in der Regel von **U** benutzt.
- In der Regel für Blöcke (**B**) kann das Attribut a definiert werden: ihm werden alle lokal (in n) aufgesammelten Deklarationen zugewiesen. Zum Aufsammeln aler lokal gültigen Deklarationen im Attribut v muss die Operation nest jetzt auf das Attribut a des umschließenden Blocks angewendet werden.
- Im Listenknoten werden die Attribute a auf alle Kinder verteilt.



Diese Attributgrammatik lässt sich nicht mehr in einer einfachern Traversierung des abstrakten Syntaxbaumes bewerkstelligen: Alle Knoten müssen zweimal besucht werden:

1. Zunächst werden die Deklarationen aufgesammelt, wobei der Syntaxbaum zunächst noch unverändert bleibt.
2. Dann werden alle Benutzungen von Bezeichnern bearbeitet.

Genauer gesagt müssen die Knoten geschachtelt traversiert werden: Beim Programmknoten und bei jedem Block muss der Unterbaum gleich noch einmal besucht werden. Hierzu verwenden wir zwei Funktionen, collect: TAB → TAB und idfy: TAB → St.

```
idfy' :: Pr -> Pr
idfy' (P s) = P s' where s' = idfy (collect (initial) s)

collect:: TAB -> St -> TAB
collect v (U _ _  ) = n where n = v
collect v (B s    ) = unnest n where n = collect (nest v) s
collect v (D x k  ) = n where n = enter v (x,k))
collect v (L s1 s2) = n' where n  = collect v s1; n' = collect n s2

idfy:: TAB -> St -> St
idfy a (U x _  ) = U x k' where k' = def a x
idfy a (B s    ) = B s' where s' = idfy a' s; a' = collect (nest a) s
idfy a (D x k  ) = D x k
idfy a (L s1 s2) = L s1' s2' where s1' = idfy a s1; s2' = idfy a s2
```

5.4 Deklarationstabellen

Der abstrakte Datentyp TAB aus Abschnitt 5.3.1 kann ebenfalls einfach in einer funktionalen Sprache implementiert werden.

```
module DeclarationTable
where
type TAB = [[(Id,Kd)]]
type Id = String
data Kd = Unbound | ...

initial :: TAB
initial = [[("read",Proc), ("write",Proc)]]

nest, unnest :: TAB -> TAB
nest t = []:t
unnest (l:t) = t

enter :: TAB -> (Id,Kd) -> TAB
enter (l:t) (x,k)
   = if (is_local l x) then error "double def." else ((x,k):l):t
     where is_local l x = any (\y->x == (fst y)) l

def :: TAB -> Id -> Kd
def t x = if t /= [] then (snd (head d')) else Unbound
          where d' = (filter (\y->x==(fst y)) (concat t))
```

Diese geradlinige Implementierung ist aber für praktische Zwecke viel zu ineffizient. Wenn auch die Implementierungen von enter, nest und unnest "optimal" sind, ist

die Komplexität der am meisten beutzte Funktion def *linear*. Da die Anzahl der Aufrufe von def ebenfalls linear von der Programgröße abhängt, wäre der Aufwand der Deklarationsanalyse quadratisch. Das ist nicht akzeptabel.

Die Implementierung von TAB muss einen Keller (*stack*) von Mengen von Daklarationen realisieren: Jeder Block enthält eine Menge von Deklarationen, und lokalen Deklarationsmengen sind wie ein Keller organisiert. In einer (rein) funktionalen Sprache könnte man die lokalen Deklarationsmengen mit geordneten Listen oder balancierten Bäumen darstellen; damit hätte def immerhin nur noch die Komplexität $\mathcal{O}(\log n)$.

Es geht in einer imperativen Sprache mt expliziter Nutzung von Zeigern aber noch effizienter. Schließlich haben wir ja schon in der lexikalischen Analyse die Bezeichner mit einer Streuspeichertabelle (*hash table*) verschlüsselt, und diese Tabelle kann als Ausgangspunkt für die Imlementierung hergenommen werden.

Wir werden die Tabelle *gitteratig* gestalten:

- Für jeden Bezeichner x gibt es eine Zeile, in der alle Deklarationen von x stehen.
- Für jeden Block git es eine Spalte, in der die Menge der darin enthaltenen Deklarationen stehen. Ganz links ist der gerade untersuchte Block, und weiter rechts stehen umschließende Blöcke.
- Die Deklarationen eines Bezeichners sind als Keller organisiert. Die erste auf dem Keller (am mesiten links stehende) ist sichtbar, die hinteren sind verdeckt.
- Die Tabelle ist dünn besetzt und wird daher mit doppelt verketteten Listen realisiert. D.h., die Einträge der Tabelle (vom Typ Kd werden um einige Zeiger erweitert:
  1. Ein Zeiger glob verseist auf die nächte globalere Definition des Bezeichners (Reihen-Verkettung).
  2. Ein Zeiger next_local verweist auf sie nächste Vereinbarung im selben Block (Spaltenverkettung).
  3. Der Schlüssel des Bezeichners verweist auf den Anfang der Liste aller Deklarationen für x. Die horizontale Liste ist also "doppelt verkettet", so dass jeder Eintrag auf den listenanfang verweist. Dies wird benötigt, um beim Löschen einer Vereinbarung für eine Bezeichner x (in der Operation unnest) die globalere Definition von x an den Kopf der Deklarationen für x zu schreiben.

Diese Implementierung ist optimal, denn alle Operationen kommen ohne Suchen aus, und insbesondere die Operation def arbeitet in konstanter Zeit: Sie muss nur den ersten Deklarations-Eintrag in der Streuspeichertabelle zurückgeben.
Bemerkung: *Zu dieser textuellen Beschreibung gibt es in den Folien wunderbare Bilder, die ich aus Zeitgründen hier nicht alle einbauen kann.*

5.5 Typanalyse

Die Typanalyse eines Programms setzt die Deklarationsanalyse voraus, d.h. dass jede Benutzung eines Bezeichners einen Querverweis auf seine Vereinbarung hat. Ausdrücke (*expression*) sind die wichtigsten Programmstücke, in denen Typen überprüft

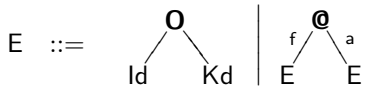
werden müssen. Deshalb beschränken wir uns hier darauf, auch wenn an anderen Programmstellen ebenfalls Typen überprüft wereden müssen.

Außerdem benutzen wir extrem vereinfachte Syntaxbäume für Ausdrücke. Jeder Ausdruck ist entweder ein Operand (ein Bezeichner) oder er ist eine Funktionsanwendung (Applikation, geschrieben $F@A$) wobei eine Funktion F auf ein Argument A angewendet wird. Sowohl F als A sind wiederum Ausdrücke. Wir betrachten also nur einstellige Funktionen; mehrstellige Funktonen lassen sich darstellen als $F@A_1@\cdots@A_k$, und binäre Operationen $\oplus$ lassen sich als $\oplus@A_1@A_2$ darstellen. Hierbei nehmen wir an, dass der Applikationsoperator "@" linksassoziativ ist, also gilt $\oplus@A_1@A_2 \equiv (\oplus@A_1)@A_2$. Unsere Funktionen haben also Typen wie $t_1 \to \cdots \to t_n \to t_0$, wo wir normalerweise einen Typen $(t_1 \times \cdots \times t_n) \to t_0$ erwarten. Nach Curry sind beide Darstellungen aber äquivalent.

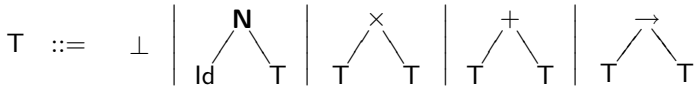
Auch bei der Typanalyse fangen zunächst mit einer einfachen Situation an, in der jede Funktion höchstens eine gültige Definition hat (Monomorphie). Danach behandeln wir den Fall, dass Funktionen *überladen* sein können, d.h., zur gleichen Zeit mehrere gültige Vereinbarungen haben können. (In den meisten Programmiersprachen sind mindestens eingebaute Operationen wie "+" überladen, z.B. für ganze und reelle Zahlen.

5.5.1 Typanalyse ohne Überladen

Die abstrakte Syntax unterscheidet nur Operandenknoten (**O**) und Applikationsknoten (**@**), wobei die Operandenknoten den **U**-Knoten nach der Deklarationsanalyse entsprechen.



Die semantische Basis umfasst einen Knotentypen (T), der folgende Definition haben könnte:



Hier stellt $\perp$ den *Fehlertyp* und **N** x t den benannten Typ x dar, dessen Vereinbarung t lautet.

Zur Typüberprüfung gibt es zwei Funktionen:

- type: Kd $\to$ T extrahiert die Typinformation aus einer Deklaration.
- check: T $\times$ T $\to$ T überprüft die Typen einer Applikation, also ob der erste Typ ein Funktionstyp ist, der auf den zweiten anwendbar ist, und liefert den resulttattyp dieser Funktion.

Die Typanalyse arbeitet *bottom-up* im Baum.