7. Abbildungsproblematik in der Computerlinguistik: Linguistische Elemente und Operationen vs. Daten- und Programmstrukturen

1. Einleitung

In dem vorliegenden Beitrag wird die Problematik der Zuordnung zwischen linguistisch orientierten Objekten und Prozessen einerseits und Objekten der Informatik andererseits anhand einiger Fallstudien erörtert. Bei Untersuchungen grundsätzlicher Art ist zunächst eine angemessene Abstraktionsebene zu bestimmen. In diesem Fall hat man sich dafür entschieden, möglichst konkret zu argumentieren, also eine niedrige Abstraktionsebene zu wählen. Als Grund hierfür ist geltend zu machen, daß im Bereich der Programmiermethodik eine Betrachtung ohne Bezug zu konkreten Beispielen erfahrungsgemäß auch den Bezug zum gesamten Gebiet der Anwendung verliert und dann lediglich Aussagen über beliebige Programme zuläßt (Stetter 1981, 133).

Mit der Auswertung der Fallbeispiele wird versucht, eine Grundthese zu belegen. Die These ist im wesentlichen negativ. Sie besagt, daß die vollständige Kenntnis eines Anwendungsproblems nicht ausreicht, um die günstigste Form der Abbildung vorgegebener Konzepte auf solche der Implementierungsebene festlegen zu können. Es bedarf hingegen nach Festlegung der Anwendungsbedingungen umfangreicher implementierungs-bezogener Überlegungen. Etwas krasser ausgedrückt: Ein Übersetzungssystem, das Spezifikationen (Booch 1983, 322 ff.) durchweg in ökonomische Lösungen transformiert, ist nicht zu erwarten. Die hier angestellten Betrachtungen bemühen sich um Berücksichtigung verschiedener Alternativen zu den üblicherweise eingesetzten Verfahren. Fragen, die sich auf die prinzipielle Realisierbarkeit von Programmierprojekten beziehen — etwa bei geringem Speicherplatzangebot oder geringer Prozessorleistung — werden hier jedoch ausgeklammert (Baase 1978, 20 f.).

Ein solcher Übersichtsartikel wäre ohne eine kritische Beurteilung der zu erwartenden zukünftigen Entwicklung auf dem betrachteten Gebiet sicherlich unvollständig. Deshalb soll am Schluß des Beitrages noch auf aktuelle Fragen der prädikativen Programmierung und die damit zusammenhängenden Aspekte der Erstellung von Expertensystemen eingegangen werden (Keen/Williams 1985, 13 ff.). Die letztgenannten Fragen reichen über die „Abbildungsproblematik“ hinaus. In gewisser Weise fordern sie zur Festlegung grundlegender Paradigmen der Programmierung für ganze Anwendungsgebiete heraus.

2. Zur Darstellung von Zeichenketten

Zeichenketten (Strings) gehören zu den elementaren Objekten der linguistisch orientierten Programmierung. Sie nehmen hier eine herausragende Stellung ein. Bei der Programmierung eines linguistischen Projekts ist die Wahl der speziellen Programmiersprache gerade im Hinblick auf die in der Programmiersprache möglichen Arten der Darstellung elementarer Daten wichtig. Wir wollen im Rahmen dieses Artikels jedoch keine solchen Aspekte der Abbildungsproblematik ausführlich erörtern, die sich unmittelbar auf die Wahl besonderer Programmiersprachen beziehen, da diesem Thema an anderer Stelle innerhalb des Buches gebührender Raum zukommt (vgl. Art. 63).

Wenige Andeutungen recht allgemeiner Art sollen an dieser Stelle genügen. Das Vorhandensein einer Programmiersprache — etwa im Unterschied zur ausschließlichen Programmierbarkeit einer Rechenanlage in der zugehörigen Maschinensprache — hat zur Folge, daß der Prozeß der Abbildung linguistischer Elemente und Operationen auf Daten- und Programmstrukturen in mindestens zwei Teilabbildungen zerlegt werden muß, wobei das Ziel einer Abbildung zur Quelle einer anderen Abbildung wird. Der Benutzer der Programmiersprache nimmt eine Abbildung des vorhandenen Sachverhaltes im Anwendungsbereich auf Begriffe und Verknüpfungen der Sprache vor. Das Übersetzungsprogramm (Compiler) bildet diese Begriffe und Verknüpfungen schließlich auf solche der Maschine ab (Schneider 1975, 61 ff.)

Dem Benutzer werden einerseits Mühen abgenommen; andererseits wird seine Freiheit zur Gestaltung des maschinellen Ablaufs erheblich eingeschränkt. Linguistisch orientierte Programmiersprachen zeichnen sich dadurch aus, daß sie die Schnittstelle der beiden genannten Abbildungen in einer vom linguistisch orientierten Benutzer als sinnvoll und hilfreich empfundenen Form festlegen. Die Sprache COMSKEE (Bertsch/Mueller-von Brochowski 1979, 1 ff.) gestattet die programmiersprachliche Behandlung von Zeichenketten einerseits in der Weise, daß letztere als untrennbare Einheiten behandelt werden, andererseits mit der Möglichkeit des lesenden und schreibenden Zugriffs auf Teilketten und einzelne Zeichen. Dies hat äußerst wichtige Konsequenzen. In gewissen Fällen wird nämlich die Zeichenkette — völlig unabhängig von ihrer Länge und Struktur — zu einem elementaren Teil des Zieles der Abbildung. In anderen Fällen wäre eine solche Darstellung weniger nützlich, weswegen ja gerade die umfangreichen Zugriffsarten auf Teile von Strings vorgesehen sind.

Abbildungsfragen unterhalb der Programmiersprache gehören zum Bereich des Compilerbaus. Im Falle dynamischer (also zur Programmerstellungszeit nicht mit maximalen Längen versehener) Zeichenketten stellt sich hier die Frage der Verwaltung von mancherlei Verweisen. Um ein verhältnismäßig kompliziertes System von Hin- und Rückverweisen handelt es sich jedenfalls dann, wenn Strings im Speicher zum Schließen von Lücken verschoben werden sollen (Hansen 1969, 499 ff.). Die Verweisstruktur ist nicht unmittelbar vom linguistischen Sachverhalt her einsichtig, sondern ergibt sich aus der Notwendigkeit häufiger Platzwechsel innerhalb eines vorgegebenen Speichers. Überhaupt ist die eindeutige Zuordnung von Teilen der Quelle einer Abbildung zu Teilen des Zieles wohl dann besonders problematisch, wenn Geschwindigkeitsfragen bei der Planung eines konkreten Algorithmus Eingang finden.

Als besonders deutliches Beispiel hierfür soll der Algorithmus von Knuth, Morris, Pratt zum Vergleich zweier Zeichenketten vorgestellt werden (Baase 1978, 173 ff.). Die gestellte Aufgabe ist die Suche nach einem vorgegebenen String P innerhalb eines längeren Strings S. Bei naiver Programmierung würden die einzelnen Zeichen $P_1, P_2, ..., P_m$ der Reihe nach mit den Zeichen $S_{i+1}, S_{i+2}, ..., S_{i+m}$ verglichen, wobei der Index i seinerseits nacheinander die Werte 0, 1, 2, ..., n—m annehmen kann. Der Vergleich von $P_1, P_2, ...$ würde jeweils dann abgebrochen, wenn sich zu einem Zeichen eine Ungleichheit ergibt. Der gesamte Vorgang könnte vorzeitig beendet werden, wenn P gefunden wird. In einem (leicht zu konstruierenden) schlechtesten Fall wäre die Anzahl der Vergleiche jedoch insgesamt etwa $n \cdot m$. Sollte m in der Größenordnung von einigen Dutzend und n in der Größenordnung von einigen Tausend liegen, wäre die Laufzeit für mehrere Vergleiche sicher nicht vernachlässigbar.

Die Idee des KMP-Algorithmus (Knuth, Morris, Pratt) liegt darin, durch eine insgesamt wenig zeitaufwendige Vorverarbeitung des Musters P jegliche Rücksprünge innerhalb des Strings S auszuschließen und damit zu einer Laufzeit von etwa n Schritten (Vergleichen und einigen Hilfsoperationen) zu kommen. Die Vermeidung des Rücksprungs beruht darauf, daß beim Scheitern eines Vergleiches nach Betrachtung von $P_1, ..., P_k$ ($k < m$) jeweils feststeht, ein wie großes Anfangsstück von $P_1, ..., P_k$ inzwischen auch als Endstück von $P_1, ..., P_k$ gelesen wurde. Somit braucht der nächste Vergleich nicht bei P_1 anzufangen, sondern hinter dem so gewonnenen Anfangsstück (etwa $P_1, ..., P_q$ für $q < k$).

Die Vorverarbeitung von P besteht in der Erzeugung eines Feldes FVERW mit m Einträgen, die zu jeder Fehlerposition in P die Position für weitere Vergleiche angeben. Dazu wird, wie man leicht einsieht, der String P an verschiedenen Positionen mit sich selbst verglichen:

```
FVERW[1]: = 0;
i: = 2;
while i < = m do
begin j : = FVERW[i—1];
      while (j < > 0) and (P[j] < > P[i—1])
            do j: = FVERW[j];
      FVERW[i]: = j + 1;
      i: = i + 1
end
```

Auf einen strengen Korrektheitsbeweis kann hier aus Platzgründen verzichtet werden.

Erforderlich ist nun weiterhin ein Verfahren, das die FVERW-Einträge während eines aktuellen Vergleichsvorgangs auswertet. Es ist dadurch gekennzeichnet, daß der Index i, der innerhalb des Strings S die Positionen 1 bis n bezeichnen kann, niemals zurückgesetzt wird:

```
i  : = 1; j: = 1;
   while i < = n do
   begin while (j < >) and (P[j] < > S[i])
         do j: = FVERW[j];
         if j = m then return "gefunden"
         else
         begin i: = i + 1; j: = j + 1 end;
   end;
   return "icht gefunden";
```

Übrigens wurde der KMP-Algorithmus, der ja m Speicherplätze für FVERW benötigt, von Galil und Seiferas (Galil/Seiferas 1983, 280 ff.) noch dahingehend verbessert, daß lediglich eine konstante, von der Länge von P unabhängige Anzahl von Speicherzellen erforderlich wird. Das Verfahren läßt sich in seiner neuen Form allerdings nicht mehr mit wenigen Zeilen beschreiben.

Inwieweit lassen sich Aussagen zur Abbildung eines Anwendungszusammenhangs auf Programm- und Datenstrukturen anhand des KMP-Algorithmus gewinnen? Die Idee des FVERW-Feldes entspringt sicherlich dem Willen, möglichst wenige überflüssige Aktionen innerhalb der Ausführung eines Programms zu dulden. Form und Funktion des FVERW-Feldes bringen jedoch keine Merkmale der Anwendung in einer von dieser her verständlichen Weise zum Ausdruck. In diesem Artikel sollen typische Zuordnungsmethoden zwischen Anwendungsgesichtspunkten (speziell linguistischer Art) und Implementierungsgesichtspunkten betrachtet werden. Was man beim KMP-Algorithmus und manchen ähnlichen Verfahren erkennen kann, ist die Auswahl bestimmter Datenstrukturen und entsprechender Verarbeitungsvorschriften, die der Anwendung unter strengen Randbedingungen hinsichtlich Zeit- und Platzbedarf gerecht werden. Weitergehende Entsprechungen sind unseres Erachtens nicht aufzeigbar.

3. Zur Abbildung von Baumstrukturen

Die syntaktische Struktur von einzelnen Zeichenketten wird sowohl bei natürlichen wie auch bei künstlichen Sprachen üblicherweise durch Bäume dargestellt (Baase 1978, 116 ff.). Für den Leser eines Buches oder den Hörer einer Vorlesung genügt zum Verständnis des Sachverhalts die bildliche Gestalt des Baumes, die aus Strichen zwischen einzelnen Punkten besteht. Der „Konsument" einer solchen Baumdarstellung befaßt sich zunächst nicht damit, welche möglichen Bewegungen innerhalb des Baumes und welche eventuellen Veränderungen vorausgesetzt werden.

Bei der Abbildung von Baumkonzepten auf konkrete Datenstrukturen sind diese Gesichtspunkte jedoch ganz entscheidend für die Wahl einer bestimmten Datenstruktur. Grundsätzlich läßt sich sagen, daß die Bereitstellung vielfältiger Operationen auf Bäumen eine entsprechende Vielfalt in der Verkettung (Verzeigerung) der Knoten erfordert. In diesem Abschnitt sollen für eine vorgegebene Baumstruktur interne Darstellungen mit einfacher, zweifacher und dreifacher Verkettung (Wedekind 1970, 199 ff.) betrachtet werden. Wir wählen den folgenden Baum als durchgängiges Beispiel:

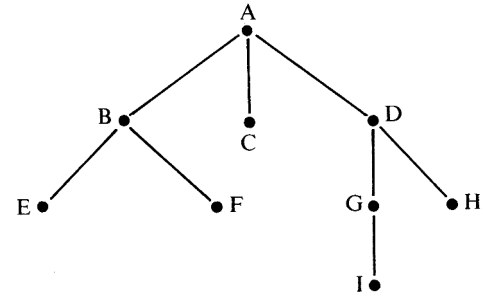

Abb. 7.1: Betrachtete Baumstruktur

Eine einfache Verkettung erfordert neben der Nutzinformation an den einzelnen Knoten (hier die Buchstaben A, B, ..., I) lediglich einen Weiterverweis. Dieser zeigt auf den im Bild rechts von dem jeweiligen Knoten stehenden Nachbarknoten. Der im Bild am weitesten links stehende abhängige Knoten (Sohn-Knoten) wird unmittelbar hinter dem Vater-Knoten gespeichert. Zu beachten sind die drei Sonderfälle, daß ein Knoten (a) einen rechten Nachbarn, aber keinen Sohn hat, daß er (b) einen Sohn, aber keinen rechten Nachbarn hat, und daß er (c) weder Nachbarn noch Sohn hat.

Fall (a) ist daran erkennbar, daß der Nachbarverweis auf die unmittelbar folgende Speicheradresse zeigt. Fälle (b) und (c) lassen sich am günstigsten durch nicht existente Verweise, etwa durch negative Adressen kennzeichnen. Wir wollen hier einfach die Buchstaben „b" und „c" als Marken vorsehen. Der obige Baum erhält somit die folgende interne Darstellung:

Adresse	Inhalt	Nachbarverweis
1	A	b
2	B	4
3	E	5
4	C	6
5	F	c
6	D	b
7	G	9
8	I	c
9	H	c

Abb. 7.2: Speicherung bei einfacher Verkettung

Die ursprüngliche bildliche Baumdarstellung läßt sich offenbar eindeutig zurückgewinnen.

Zur einfachen Verkettung in der gewählten Art gibt es eine Reihe von Alternativen, die wir nicht betrachten wollen, etwa die genau umgekehrte Behandlung von Sohn- und Nachbarverweisen oder die Kennzeichnung mit zusätzlichen Bits für die genannten Sonderfälle.

Bei der zweifachen Verkettung wird je ein Verweis für den rechten Nachbarn und für den am weitesten links stehenden Sohn bereitgestellt. Bei nicht vorhandenen Söhnen oder Nachbarn wird ein Leerverweis (*nil*) gesetzt. Wir erhalten damit

Adresse	Inhalt	Nachbar-verweis	Sohnverweis
1	A	nil	2
2	B	3	5
3	C	4	nil
4	D	nil	7
5	E	6	nil
6	F	nil	nil
7	G	8	9
8	H	nil	nil
9	I	nil	nil

Abb. 7.3: Speicherung bei zweifacher Verkettung

Wir kommen zur dreifachen Verkettung von Knoten. Hierbei wird neben den Angaben, die zur zweifachen Verkettung gehören, noch ein Vaterverweis vorgesehen. Die Darstellung ergibt sich in naheliegender Weise.

Allen betrachteten Tabellen ist gemeinsam, daß sie neun Baumknoten innerhalb eines zusammenhängenden Speicherabschnittes mit Inhalten und ausreichender Information über deren strukturelle Abhängigkeit darstellen. Die Tabellen unterscheiden sich in der Ausführlichkeit der Angaben. Die Ausführlichkeit läßt sich wiederum über die vorgesehene Verwendung begründen. Wenn während der Laufzeit eines Programmes Veränderungen von Baumstrukturen vorgesehen sind, ist die einfache Verkettung unzureichend. Jedes Einfügen, Löschen oder Umhängen müßte von einer Neuordnung des Baumes begleitet sein, da Sohnknoten grundsätzlich unmittelbar hinter dem zugehörigen Vaterknoten abgelegt sind. Eine Bewegung von Zeigern innerhalb des Baumes wäre weiterhin nur in der Richtung von der Wurzel zu den einzelnen Knoten möglich.

Die zweifache Verkettung gestattet ein einfaches Einfügen, Löschen und Umhängen. Hierbei wollen wir von der gesonderten Problematik der Ausnutzung ungültig gewordener Speicherplätze einmal abgesehen. Auch hier ist jedoch keine Zeigerbewegung zur Wurzel hin möglich, da alle Verweise auf Nachbarn oder Söhne zeigen.

Die dreifache Verkettung löst schließlich auch dieses Problem. Über den Vaterverweis kann in einem oder mehreren Schritten die Wurzel des Teilbaumes und die Wurzel des Gesamtbaumes erreicht werden. Bei der Auswahl einer geeigneten Verkettungstechnik für eine bestimmte, an linguistischen Sachverhalten orientierte Anwendungsaufgabe wirken die dem Anwendungsproblem inhärenten Bedingungen und die der Informatik entstammenden Effizienzgesichtspunkte zusammen. In manchen Fällen ist eine Wahl effizienter Verfahren unabdingbar; in anderen Fällen lassen sich auch zeit- und platzaufwendige Prozeduren vertreten. Hier sei auch angemerkt, daß die ständigen Verbesserungen der Rechnertechnologie auf der Gatter-Ebene und der Register-Transfer-Ebene den Anwendern manche Probleme abnehmen. Allerdings werden die Aufgaben ebenfalls anspruchsvoller, und bei Großprojekten kann auf wirtschaftlichen Einsatz der verfügbaren Mittel nicht verzichtet werden.

Entsprechend den recht unterschiedlichen und teilweise widersprüchlichen Anforderungen an Baumkonzepte in linguistischen und anderen Problemkreisen hat sich bislang

keine allgemein gültige Baumdefinition auf der Programmiersprachenebene durchgesetzt. Allenfalls bei Verbunden (Wirth 1982, 76 ff.) kann man in eingeschränktem Sinn von Bäumen sprechen, deren Aufbau jedoch während eines Programmlaufs konstant bleiben muß. LISP bietet über die CAR- und CDR-Operationen eine einfache Form der Verweisverwaltung an (Winston/Horn 1981, 18 ff.). Eine frühere Version der Sprache COMSKEE enthielt eine an LISP angelehnte Form der Baumverwaltung. Allerdings ging deren Einsatzbereich nur unwesentlich über andersartige Möglichkeiten innerhalb derselben Sprache hinaus, so daß bei späteren Sprachversionen auf ein vorgegebenes Baumkonzept verzichtet wurde. Eine bedeutende Rolle spielen Baumkonzepte auch unterhalb der Programmiersprachen-Ebene, und zwar zur Verwaltung von größeren Datenbeständen, die während eines Programmlaufs erweitert oder anderweitig geändert werden müssen.

4. Syntaktische Analyseverfahren

Zu den Objekten, die in nahezu jedem System zur automatischen Verarbeitung linguistischer Daten einer konkreten Darstellung bedürfen, gehören kontextfreie Grammatiken (Chomsky 1959, 137 ff.). Zwar erfordert eine angemessene Berücksichtigung natürlichsprachlicher Zusammenhänge immer auch den Rückgriff auf Kontext-Sensitivitäten unterschiedlicher Art; jedoch wird das Konzept der kontextsensitiven Grammatiken nicht in der im Rahmen der theoretischen Informatik üblichen Allgemeinheit verwendet (Maurer 1969, 28).

Sachverhalte, die sich nicht vollständig durch kontextfreie Regelmengen darstellen lassen, werden üblicherweise durch Zusätze zum Grundkonzept der kontextfreien Grammatik erfaßt (Dowty/Karttunen/Zwicky 1985, 3 ff.). Die Bedeutung dieser Vorgehensweise darf nicht unterschätzt werden. Nur auf diese Weise kann nämlich gewährleistet werden, daß strukturellen Abhängigkeiten mit Hilfe von Ableitungsbäumen eine verbildlichte Ausdrucksform zukommt. Anders ausgedrückt: Würden allgemeine kontextsensitive Regelmengen eingesetzt, so entfiele — von speziellen Ausnahmen abgesehen — die 'Intuition des Baumes' als Mittel der Darstellung.

In diesem Abschnitt sollen ausschließlich kontextfreie Regelmengen betrachtet werden, diese jedoch in voller Allgemeinheit. Damit wird auch ausgedrückt, daß speziellere Klassen von Grammatiken unseres Erachtens keine ausgiebige Berücksichtigung im Rahmen der linguistischen Datenverarbeitung verdienen. Die zur Analyse künstlicher Sprachen üblichen Grammatiktypen, insbesondere LL(1)- und LALR(1)-Grammatiken (Knuth 1971, 79 ff.), setzen neben der Eindeutigkeit aller zu analysierender Sätze eine völlige Determiniertheit der weiteren Analyse aufgrund eines bereits gelesenen Teilsatzes voraus. Diese Eigenschaft ist jedoch wegen der beträchtlichen Homonymie und strukturellen Mehrdeutigkeit natürlicher Sprachen geradezu abwegig, wenn mehr als ein künstlich eingeschränkter Teilbereich natürlichsprachlicher Äußerungen erfaßt werden soll.

Zur effizienten Analyse anhand kontextfreier Grammatiken soll das weithin bekannte Analyseverfahren von Cocke, Kasami, Younger (Younger 1967, 189 ff.) vorgestellt werden. Dieses stand längere Zeit neben einem scheinbar völlig anderen Verfahren von Earley (Earley 1970 a, 94 ff.), das einer wesentlich komplizierteren Erläuterung bedürfte. In der neueren Literatur (Graham/Harrison/Ruzzo 1980, 415 ff.) wurde jedoch deutlich herausgearbeitet, daß beide Verfahren in naheliegender Weise als spezielle Varianten einer einzigen Prozedur anzusehen sind. Man wählt hier das CKY-Verfahren (Cocke, Kasami, Younger), weil es in recht knapper Form beschrieben werden kann. Zunächst sind einige Vorbereitungen nötig: eine kontextfreie Grammatik $G = (N, \Sigma, P, S)$ mit Nichtterminalalphabet N, Terminalalphabet Σ, Regelmenge P und Startsymbol S heißt Chomsky-Normalform-Grammatik, wenn alle Produktionen von der Form $A \rightarrow BC$ für A, B, C, $\in N$ oder von der Form $A \rightarrow a$ für $A \in N$, $a \in \Sigma$ sind. Der CKY-Algorithmus gewinnt seine Einfachheit weitgehend aus der Tatsache, daß er Chomsky-Normalform-Grammatiken voraussetzt. Es ist jedoch unmittelbar einzusehen, daß hierin keine Beschränkung der Allgemeinheit liegt. Insbesondere werden alle Regeln $A \rightarrow B_1 \ldots B_n$ mit $n > 2$ ersetzt durch neue Regelmengen der Gestalt $A \rightarrow B_1 C_1$, $C_1 \rightarrow B_2 C_2$, ..., $C_{n-3} \rightarrow B_{n-2} C_{n-2}$, $C_{n-2} \rightarrow B_{n-1} B_n$, die ihrerseits den Bedingungen der Normalform genügen. Sei nun das zu analysierende Wort (im linguistischen Sprachgebrauch handelt es sich bei einer solchen Analyse wohl überwiegend um einen Satz oder ein Gebilde aus

mehreren Sätzen) $w = a_1 a_2 ... a_n \in \Sigma^+$ mit $a_i \in \Sigma$ für $i \in \{1, ..., n\}$. Der CKY-Algorithmus liefert im wesentlichen eine Dreiecksmatrix T mit Einträgen t_{ij} für $i \in \{1, ..., n\}$, $j \in \{1, ..., n - i + 1\}$.

Jedes t_{ij} ist eine Teilmenge von N mit der Eigenschaft: $A \in t_{ij}$ genau dann, wenn $a_i a_{i+1} ... a_{i+j-1}$ mit Hilfe der Regeln aus P von A abgeleitet werden kann. Insbesondere ist $S \in t_{1n}$ genau dann, wenn w zur Sprache gehört ($w \in L(G)$).

Man kann die Prozedur folgendermaßen skizzieren:

(i) j: = 1;
(ii) i: = 1;
(iii) a) Für j = 1 ist

$$t_{ij} = \{A \in N \mid A \rightarrow a; \in P\}.$$

b) Für $1 < j \leq n - i + 1$ ist

$$t_{ij} = \{A \in N \mid \text{Es gibt } k \in \{1, ..., j - 1\}, B \in t_{ik} \text{ und } C \in t_{i+k,j-k} \text{ mit } A \rightarrow BC \in P\}.$$

c) Für $j > n - i + 1$ ist die Matrix völlig konstruiert.

(iv) Erhöhe i um 1.
a) Für $i \leq n - j + 1$ fahre fort bei (iii).
b) Für $i > n - j + 1$ erhöhe j um 1 und fahre fort bei (ii).

Schritt (iii) a) ordnet jedem Terminalzeichen die nach Definition der Normalform-Grammatik vorhandenen Nichtterminalzeichen zu. Schritt (iii) b) stellt den eigentlichen Kern des Algorithmus dar. Für das jeweilige j sind k und j—k kleiner als j; folglich sind t_{ik} und $t_{i+k,j-k}$ schon zu einem früheren Zeitpunkt gebildet worden.

Beispiel: Es sei G = ({S, A}, {a, b}, P, S) mit den Produktionsregeln

(1) S → AA, (2) S → AS, (3) S → b
(4) A → SA, (5) A → AS, (6) A → a

i \ j	1	2	3	4	5
1	A	S,A	S,A	S,A	S,A
2	S	A	S	S,A	
3	A	S	S,A		
4	A	S,A			
5	S				

Abb. 7.4: Analysematrix

(Die Numerierung der Regeln wird erst später verwendet.) Man erhält folgende Analysematrix für das Wort w = abaab: (s. Abb. 7.4).

Da $S \in t_{1,5}$ ist, gehört abaab zur Sprache. Wenn die Analyse zu einem Ableitungsbaum führen soll, muß die Matrix weiter ausgewertet werden. Die Spezifikation dieser Teilaufgabe kann man so formulieren, daß jeweils zu einem festen Nichtterminalzeichen U und zwei Positionen i und j im Eingabewort ein Baum Baum(i, j, U) gefunden werden möge, der eine Ableitung von U zu $a_i a_{i+1} ... a_{i+j-1}$ darstellt. Der Baum sei zunächst als eine Folge von Regelnummern beschrieben. (Dazu ist vorauszusetzen, daß die Regelmenge in eindeutiger Weise numeriert ist.)

Man kann das Verfahren wie folgt skizzieren:

Man erhält bei der Berechnung von Baum (1, n, S) eine Darstellung des Ableitungsbaumes von $a_1 ... a_n$.

Für das Wort abaab und die oben angegebene Grammatik G geschieht folgende Berechnung:

(i) Ist j = 1 und $U \rightarrow a_i \in P$ die Produktion mit der Nummer m, dann gib m aus.

(ii) Ist $j > 1$ und $k \in \{1, ..., j - 1\}$ die kleinste Zahl, für die $V \in t_{ik}$ und $W \in t_{i+k,j-k}$ existieren mit $U \rightarrow VW \in P$, dann gib die Nummer einer solchen Produktion aus. Anschließend berechne (rekursiv) Baum(i, t, V) und Baum (i + k, j — k, W).

Baumfunktion	Regelnummer
Baum (1,5,S)	1
Baum (1,1,A)	6
Baum (2,4,A)	4
Baum (2,1,S)	3
Baum (3,3,A)	5
Baum (3,1,A)	6
Baum (4,2,S)	2
Baum (4,1,A)	6
Baum (5,1,S)	3

Abb. 7.5: Folge der Aktionen

Die ausgegebene Folge von Regelnummern beschreibt ihrerseits gerade (als 'Linksableitung') den Ableitungsbaum (s. Abb. 7.6).

Man hat nun zu fragen, in welcher Weise die Abbildung linguistisch orientierter Objekte und Prozesse auf Datenstrukturen und programmtechnische Anweisungen in die-

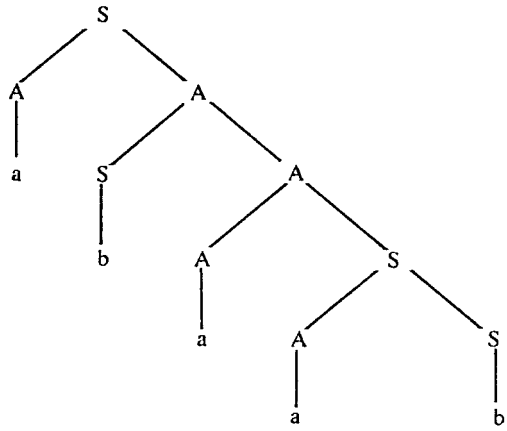

Abb. 7.6: Erzeugte Baumstruktur

sem nicht ganz trivialen Fallbeispiel erfolgt. Für den Linguisten umfaßt der Prozeß der syntaktischen Analyse die Eingabe einer Zeichenkette, die Bereitstellung einer Regelmenge und die Ausgabe eines Ableitungsbaumes, der die Zeichenkette zu der Regelmenge in Beziehung setzt. Die drei genannten Objekte finden sich auch innerhalb der betrachteten programmtechnischen Darstellung, wobei über die genaue Form der Codierung der Regelmenge nichts ausgesagt wurde. Die Beschreibung des Baumes durch eine Zahlenfolge mag einen ungeübten Leser zunächst erstaunen. Es läßt sich jedoch leicht einsehen, daß Äquivalenz vorliegt. Die Problematik der Abbildung zeigt sich deutlich im Konzept der Analysematrix, das durch keinerlei linguistische Sachverhalte motiviert erscheint.

Die Idee der Analysematrix ist mit Effizienz-Überlegungen verbunden. Jeder Eintrag der Matrix repräsentiert die Analyse für einen möglichen Teilstring der Eingabe-Zeichenkette. Zeilen und Spalten entsprechen gerade den möglichen Anfangs- und Endpositionen solcher Teilstrings. Die Matrizen-Schreibweise gestattet weiterhin eine elegante Formulierung der Zugriffe auf Teilergebnisse der Analyse. Man kommt zu dem Ergebnis, daß die Matrix keine Entsprechung im vorgegebenen linguistischen Sachverhalt besitzt und daß keine einzelne Komponente der linguistischen Fragestellung auf das Konzept 'Analysematrix' abgebildet wird. Die Matrix ist eine Hilfskonstruktion, deren Bedeutung in der Unterstützung anderer programmtechnischer Komponenten liegt. Der betrachtete Fall stützt die Auffassung, daß eine systematische Methodenlehre der Abbildung von Anwendungsgesichtspunkten auf Informatik-Konzepte wohl nicht erreicht werden kann. Mit anderen Worten: Die Analogie der linguistischen Objekte und Operationen zu den in einer konkreten Implementierung vorhandenen Objekten und Operationen bleibt nach unserer Auffassung recht oberflächlich. Eventuelle Effizienzgesichtspunkte der Rechner-Darstellung lassen sich nicht schon aus inhärenten Kriterien des Anwendungsfalls ableiten.

5. Ausblick

In der Informatik zeichnet sich derzeit ein deutlicher Trend zur nichtprozeduralen Programmierung ab (Bertsch 1985, 137 ff.). Unter diesem Begriff sind mehrere Ansätze zusammengefaßt, deren einflußreichster im Umfeld der Programmiersprache PROLOG (Shapiro/Takeuchi 1983, 25 ff.) und der neueren Expertensysteme zu erkennen ist. Der Einfluß der von PROLOG angestoßenen Bewegung zur prädikativen (genauer: prädikatenlogischen) Darstellung von Abläufen geht erstens auf die Forderung nach präzisen Korrektheitsbeweisen zurück, zweitens auf die zunehmende Bedeutung von Methoden für die industrielle Software-Erstellung, die früher nur innerhalb von Projekten der Künstlichen Intelligenz verwendet wurden, drittens sicherlich auf die Ankündigung der japanischen Computer-Industrie, bei der nächsten Generation ihrer Rechner-Architekturen bevorzugt prädikative (PROLOG-ähnliche) Verarbeitungsformen zu berücksichtigen (Shapiro 1983, 1 ff.). Ob sich die Hoffnungen, auf diese Weise eine bis zu tausendfache Beschleunigung gegenüber heutigen KI-Systemen zu erreichen, erfüllen werden, kann man zum gegenwärtigen Zeitpunkt (1987) in keiner Weise beurteilen.

Es ist zu vermuten, daß eine so starke Tendenz innerhalb der Kerninformatik in kürzester Zeit auch in den Anwendungsgebieten Eingang finden wird. Für den linguistischen Bereich dürfte dies heißen, daß nicht nur Grammatiken und Wörterbücher, sondern auch die Vorschriften zur Bearbeitung derselben überwiegend nichtprozedural formuliert werden. Es gibt keinen Grund zu der Annahme, daß die Prädikatenlogik hier weniger geeignet sei als in anderen Disziplinen, universelle Abläufe zu beschreiben.

Beim Einsatz grundsätzlich anderer Programmier-Paradigmen wird allerdings auch

die Abbildungs-Problematik in grundsätzlich anderer Weise aktuell. Üblicherweise kann bei prädikativer Programmierung nicht zwischen Daten und Anweisungen unterschieden werden. Beide Objektklassen bedürfen der Formulierung von Aussagen über sie und werden durch diese definiert. Beispielsweise ist es mit gewissen Varianten von PROLOG möglich, Datenbanken und die zugehörigen Manipulationsverfahren mit einem gemeinsamen Anweisungsformat zu spezifizieren.

Deshalb ist zum Abschluß des Beitrages einschränkend zu konstatieren, daß die vorliegenden Erörterungen zunächst für eine linguistische Datenverarbeitung auf der Grundlage prozeduraler Programmierung Gültigkeit haben. Zu den prozeduralen Sprachen zählen etwa FORTRAN, PASCAL, COMSKEE und ADA sowie die überwiegende Mehrzahl der sonstigen gebräuchlichen Sprachen. Für nichtprozedurale Sprachen und Systeme gelten andere Kriterien, die vermutlich in den nächsten Jahren Gegenstand der weiteren Forschung sein werden. Mit dem Hinweis auf die Aktualität dieser Fragen soll allerdings nicht angedeutet werden, daß sich das prädikative Paradigma in einem bestimmten Anwendungsbereich auf Dauer als einziges durchsetzen könne. Es dürfte eher zu einer fruchtbaren Debatte um Vor- und Nachteile der unterschiedlichen Ansätze kommen, als deren Resultat bestimmten Teilen der prozeduralen Programmierung anhaltende Bedeutung zuerkannt werden muß.

6. Literatur (in Auswahl)

S. Baase 1978 · E. Bertsch 1985 · E. Bertsch/A. Mueller-von Brochowski 1979 · G. Booch 1983 · N. Chomsky 1959 · D. R. Dowty/L. Karttunen/A. M. Zwicky 1985 · J. Earley 1970 a · Z. Galil/J. Seiferas 1983 · S. L. Graham/M. A. Harrison/W. L. Ruzzo 1980 · W. Hansen 1969 · M. J. R. Keen/G. Williams 1985 · D. E. Knuth 1971 · H. Maurer 1969 · E. Y. Shapiro 1983 · E. Y. Shapiro/A. Takeuchi 1983 · H. J. Schneider 1975 · F. Stetter 1981 · H. Wedekind 1970 · P. H. Winston/B. K. P. Horn 1981 · N. Wirth 1982 · D. H. Younger 1967.

Eberhard Bertsch, Bochum
(Bundesrepublik Deutschland)

8. Application and Research — Mutual Side Effects

1. Research, Application and Honesty

Many Research and Development projects in natural language processing are improductive: they are not ventursome enough to yield new knowledge, and the knowledge they build on is not established safely enough to give robust tools which can be used for practical purposes. When attacked because a project produces only trivial knowledge, the planners refer to production needs: After all, they say, we have users to serve and cannot play around irresponsibly. And when users complain that the system does not perform properly, they are dismissed with the remark that building up knowhow is more important than immediate production. Probably this risk for risk evasion is inherent in the very concept of Research and Development. It would be much more appropri-