```
(dolist (y (get-categories feld-2))
  (setf regeln1
        (remove-if #'(lambda (z) (not (equal (rest z) (list x y)))) regeln))
  (when regeln1
    (setf resultat (append resultat
                           (mapcar #'(lambda (z)
                                       ;; Generierung eines Eintrags:
                                       (list (first z)
                                             (cons x (first feld-1))
                                             (cons y (first feld-2))))
                                   regeln)))))))
```

```
;;; GET-CATEGORIES
;;; Diese Funktion liefert als Wert die Liste aller Kategorien, für die es einen Eintrag
;;; in FELD gibt.
(defun get-categories (feld)
  (if (symbolp (first feld)) feld (make_set (mapcar #'first feld))))

;;; MAKE-SET
;;; Alle Vorkommen eines Symbols in einer Liste außer dem letzten werden entfernt.
(defun make_set (set &key (test #'eq))
  (cond ((endp set) nil)
        ((member (first set) (rest set) :test test) (make_set (rest set)))
        (t (cons (first set) (make_set (rest set))))))
```

Generierung von Strukturbeschreibungen

Um aus den Recognizern Parser zu erhalten, ist es ausreichend, die im letzten Abschnitt angegebenen Programme durch eine Funktion zu ergänzen, die aus der Chart einen Leftparse bzw. eine Strukturbeschreibung generiert.

```
;;; PARSE-CKY
;;; Hauptfunktion.
(defun parse-cky (satz &optional (symbol (startsymbol)))
  (let ((chart (make-chart satz)))
    (pprint chart) (terpri)
    (if (assoc symbol (first chart))
        ;; an dieser Stelle ist ein Aufruf der Funktion einzufügen, die aus der Chart einen
        ;; Leftparse bzw. eine Strukturbeschreibung erzeugt:
        ;; { (STRUKTUR ...) | (LEFT-PARSE ...) }
        (format t "Der Satz ist nicht wohlgeformt.~%"))))
```

```
;;; INDEX
;;; Diese Funktion berechnet den Index von REGEL.


(defun index (regel regel-set)
  (+ (- (length regel-set) (length (member regel regel-set :test #'equal))) 1))


;;; LEFT-PARSE
;;; Diese Funktion generiert aus der einfachen Analysematrix CHART einen Leftparse
;;; für SATZ. Aufruf: (left-parse (length satz) 1 symbol satz chart).


(defun left-parse (i j kat satz chart)
  (if (= i (- j 1)) (print (nth (- j 1) satz))
      (let ((seiten (expansionen kat)))
        (dolist (x seiten)
          (do ((k (+ i 1) (+ k 1))) ((eq k j))
              (when (and (member (first x) (assoc '(,i ,k) chart :test #'equal))
                         (member (second x) (assoc '(,k ,j) chart :test #'equal)))
                (print (index (cons kat x) (regeln)))
                (left-parse i k (first x) satz chart)
                (left-parse k j (second x) satz chart)))))))


;;; STRUKTUR
;;; Diese Funktion generiert aus der komplexen Analysematrix CHART eine Struk-
;;; turbeschreibung für SATZ. Aufruf: (struktur symbol 1 (length satz) satz chart).


(defun struktur (kat i j satz chart)
  (if (= i (- j 1)) (list kat (nth (- j 1) satz))
      (let ((eintrag (assoc kat (assoc (list i j) chart :test #'equal))))
        (list kat (struktur (first (second eintrag)) (second (second eintrag))
                            (third (second eintrag)) satz chart)
                  (struktur (first (third eintrag)) (second (third eintrag))
                            (third (third eintrag)) satz chart))))))
```

Aufgaben

6.1 Erweitern Sie den Prolog-Recognizer zu einem Parser.

6.2 Die in Kapitel 6.2 beschriebenen Verfahren zur Generierung einer Analysematrix berechnen eine Matrix, indem zunächst die Diagonale j–i = 1 berechnet wird, dann die Diagonale j–i = 2, usw. Eine andere Möglichkeit besteht darin, die Matrix spaltenweise von links nach rechts zu berechnen. Implementieren Sie diese Variante der Berechnungsverfahren.

6.3 Modifizieren Sie die Prozeduren LINKSPARSE und STRUKTUR so, daß sie für ambige Sätze alle Linksableitungen / Strukturbeschreibungen generieren.

6.4 Ergänzen Sie den Cocke-Kasami-Younger-Algorithmus mit einer Heuristik, die eine beschränkte Verarbeitung von Tilgungsregeln ermöglicht.

6.5 Entwickeln Sie auf Grundlage der in Kapitel 5 vorgestellten Algorithmen einen Chart-Parsingalgorithmus, der die Left-corner-Strategie verwendet.

Kapitel 7

Der Earley-Algorithmus

Wir beginnen dieses Kapitel mit einer kurzen Beschreibung der wichtigsten Unterschiede zwischen dem im letzten Kapitel behandelten Cocke-Kasami-Younger-Algorithmus und dem Earley-Algorithmus[1]. Anschließend führen wir die drei vom Earley-Algorithmus zur Berechnung der Chart verwendeten Prozeduren ein. Mithilfe dieser Prozeduren formulieren wir zunächst einen Erkennungsalgorithmus für kontextfreie Syntaxen ohne Tilgungs- und Kettenregeln. Dieser Algorithmus bildet die Grundlage für die Entwicklung verschiedener Varianten von Erkennungs- und Parsingalgorithmen für beliebige kontextfreie Syntaxen.

7.1 Unterschiede zwischen dem Cocke-Kasami-Younger- und dem Earley-Algorithmus

Unterschiede zwischen beiden Algorithmen gibt es sowohl hinsichtlich der verarbeitbaren Syntaxen, als auch der in der Chart gespeicherten Informationen und der Art der Berechnung der Chart:

1. Der Earley-Algorithmus erlaubt anders als der Cocke-Kasami-Younger-Algorithmus die Verarbeitung beliebiger kontextfreier Syntaxen: Die Syntax muß nicht in Chomsky-Normalform vorliegen. Sie darf neben links- und rechtsrekursiven Regeln auch Tilgungsregeln und sogar Zyklen enthalten.

2. Es werden nicht nur passive, sondern auch aktive Kanten in die Chart eingetragen.

[1] Eine Gemeinsamkeit bildet der Zeit- und Raumbedarf beider Algorithmen: wie beim Cocke-Kasami-Younger-Algorithmus ist der Zeitbedarf $O(n^3)$ und der Raumbedarf $O(n^2)$.

3. Der Cocke-Kasami-Younger-Algorithmus ist ein reiner Bottom-up-Algorithmus und weist die charakteristischen Mängel dieses Verfahrens auf: Es werden Konstituenten gebildet, ohne daß sichergestellt ist, ob sie zu größeren Konstituenten kombiniert werden können[2].

   Beispiel (7-1)

   Syntax : $\{S \rightarrow XX, S \rightarrow YB, X \rightarrow XX, X \rightarrow AB, Y \rightarrow BA, Y \rightarrow YB\}$
   Lexikon : $a \in A$, $b \in B$
   Satz : abab

[2] In [?] wird von einem Experiment berichtet, in dem 80% der von einem mit diesem Algorithmus arbeitenden Parser gebildeten Konstituenten zu diesem Typ gehörten.

Der Cocke-Kasami-Younger-Algorithmus generiert eine Chart mit folgenden Kanten:

$Chart(0, 4) = \{S, X\}$
$Chart(1, 4) = \{S, Y\}$
$Chart(0, 2) = \{X\}$
$Chart(1, 3) = \{Y\}$
$Chart(2, 4) = \{X\}$
$Chart(0, 1) = Chart(2, 3) = \{A\}$
$Chart(1, 2) = Chart(3, 4) = \{B\}$

Die Symbole in Chart(1,4) und Chart(1, 3) repräsentieren für die Analyse des Satzes nicht weiter verwendbare Konstituenten. Der Earley-Algorithmus dagegen ist kein Bottom-up-Algorithmus, sondern realisiert eine gemischte Strategie, die Aspekte des Top-down-Parsens mit denen des Bottom-up-Parsens so miteinander verbindet, daß die Bildung nicht weiter verwendbarer passiver Kanten weitgehend vermieden wird.

Ein weiterer Vorzug des Earley-Algorithmus besteht darin, daß es durch geringfügige Modifikationen möglich ist, bestimmte von Linguisten häufig verwendete Notationskonventionen effizient zu verarbeiten, durch die z.B in Regeln Kategorien als *fakultativ* markiert werden oder festgelegt wird, daß eine Kategorie an der gegebenen Position beliebig oft vorkommen kann.

Beispiel (7-2)

(1) S $\rightarrow$ NP VP {PP}	-	fakultative Kategorie PP
(2) N1 $\rightarrow$ A^*N	-	n-mal die Kategorie A ($n \geq 0$)
(3) N1 $\rightarrow$ A^+N	-	n-mal die Kategorie A ($n \geq 1$)[3]

Zur Repräsentation der Kanten werden häufig *geteilte Produktionen* verwendet:

Wenn G = $<V_N, V_T, S, R>$ eine beliebige kontextfreie Syntax ist, dann ist R'=$\{X \rightarrow \alpha.\beta \mid X \rightarrow \alpha\beta \in R$, mit $\alpha, \beta \in V^*\}$ die Menge der geteilten Produktionen von G.

In diesem Fall korrespondieren Produktionen der Form $A \rightarrow \alpha.\beta$, mit $\beta = e$, passiven Kanten und Produktionen mit $\beta \neq e$ aktiven Kanten; α repräsentiert den *geschlossenen* Teil und β den *offenen* Teil dieser Kante.

7.2 Darstellung des Erkennungsalgorithmus

Ähnlich wie beim Cocke-Kasami-Younger-Algorithmus kann auch beim Earley-Algorithmus die Chart von links nach rechts aufgebaut werden:

[3] Formal gesehen erhält man durch die Verwendung solcher Metazeichen Regelschemata, die eine (endliche/nicht-endliche) Menge von kontextfreien Regeln zusammenfassen.

Für einen Satz $w = w_1 \ldots w_n$ ($n \geq 1$) werden für die Punkte 0, 1, ..., n sukzessiv alle Kanten berechnet und in die Chart eingetragen, die bis zu diesem Punkt reichen. Werden Kanten durch geteilte Produktionen realisiert, dann repräsentiert eine Produktion $A \rightarrow \alpha.\beta \in \mathit{Chart}(i, j)$[4] eine Konstituente, deren erkannter Teil von w_{i+1} bis w_j reicht. Um diese Konstituente zu vervollständigen, muß versucht werden, einen Satzabschnitt $w_{j+1}, \ldots, w_k$ ($k \leq n$) als β zu analysieren (vgl. Abbildung 7-1).

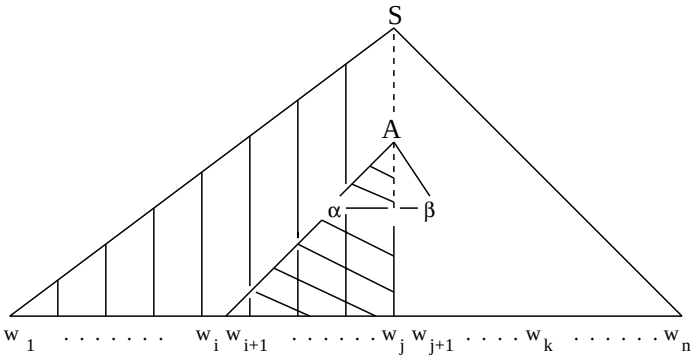


Abbildung (7-1)

Anders formuliert: $A \rightarrow \alpha.\beta \in$ Chart(i, j) gdw.

1. $S \overset{*}{\Rightarrow} w_0 \ldots w_i\ A\ \tau$, mit $\tau \in V^*$; d.h. es muß eine Ableitung in G geben, in der $w_0 \ldots w_i$ einen Präfix von $A\ \tau$ bilden, und

2. $\alpha \overset{*}{\Rightarrow} w_{i+1} \ldots w_j$; d.h. es muß möglich sein, $w_{i+1} \ldots w_j$ als α zu analysieren bzw. auf α zu reduzieren.

Da die erste Bedingung charakteristisch für Top-down-Parsen ist und die zweite für Bottom-up-Parsen, werden sie auch als „Top-down-Bedingung“ bzw. „Bottom-up-Bedingung“ bezeichnet. Die drei zur Berechnung der Chart verwendeten Prozeduren EXPAND, COMPLETE und SHIFT stellen sicher, daß alle Kanten, die in die Chart eingetragen werden, diese beiden Bedingungen erfüllen.

[4] Auch hier gibt es natürlich wieder ganz verschiedene Möglichkeiten, die Chart (und die Kanten) zu realisieren; z.B. als Matrix, die für einen Satz der Länge n ($n \geq 1$) aus (n+1) x (n+1) Feldern besteht, von denen allerdings nur die ersten i Felder ($0 \leq i \leq n$) jeder Spalte benötigt werden.

EXPAND

Durch diese Prozedur werden Kanten generiert, die Möglichkeiten darstellen, wie eine begonnene Ableitung top-down fortgesetzt werden kann. Sie trägt zyklische Kanten in die Chart ein: Ausgehend von einer bereits in der Chart enthaltenen aktiven Kante wird für jede Regel der Syntax, deren linke Seite mit dem ersten Symbol des offenen Abschnitts der Kante identisch ist, eine neue Kante in die Chart eingetragen. Tilgungsregeln führen zum Eintrag von passiven Kanten; alle übrigen Regeln zu aktiven Kanten.

PROZEDUR EXPAND

DATEN: Eine kontextfreie Syntax G = <V_N, V_T, S, R> und eine Chart C.

METHODE:
Wenn <[i, j, A, α, Bβ] $\in$ C, mit $(0 \leq i \leq j \leq n)$>
Dann <Für alle B $\rightarrow \tau \in$ R:
<Trage [j, j, B, e, τ] in C ein>>

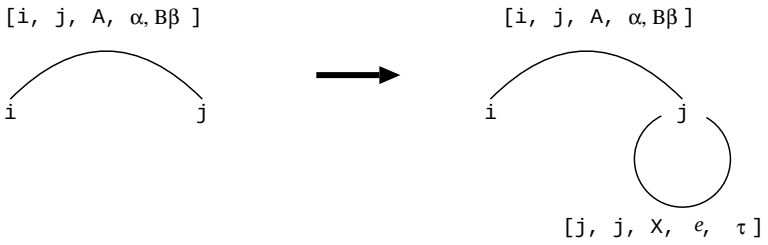


Abbildung (7-2)

COMPLETE

Durch Kombination einer aktiven Kante mit einer geeigneten passiven Kante generiert COMPLETE eine neue aktive/passive Kante und trägt sie in die Chart ein; d.h. die aktive Kante wird bottom-up durch eine passive ergänzt.

PROZEDUR COMPLETE

DATEN: Eine Chart C.

METHODE:
Wenn <[i, j, A, α, Bβ] $\in$ C und [j, k, B, τ, e] $\in$ C, mit $0 \leq i \leq j \leq k \leq n$>
Dann <Trage [i, k, A, αB, β] in C ein>

Eine aktive Kante k_a kann mit einer passiven Kante k_p kombiniert werden, wenn das erste Symbol des offenen Abschnitts von k_a mit dem Kopf von k_p übereinstimmt und $Ende(k_a) = Start(k_p)$.

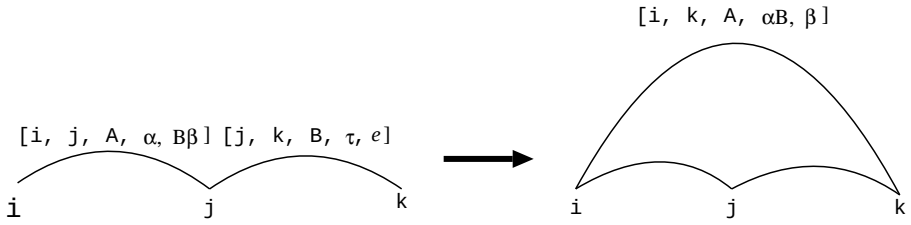


Abbildung (7-3)

SHIFT

Wenn das erste Symbol des offenen Abschnitts einer aktiven Kante ein terminales Symbol ist und mit dem nächsten Wort des Satzes übereinstimmt, wird eine Kante in die Chart eingetragen, die sich von der gegebenen Kante nur darin unterscheidet, daß das terminale Symbol das letzte Symbol des geschlossenen Abschnitts der neuen Kante bildet.

PROZEDUR SHIFT

DATEN: Ein Satz $w = w_1 \ldots w_n$, $n \geq 1$ und eine Chart C.

METHODE:

Wenn $<$ [i, j-1, A, α, $w_j\beta$] $\in$ C$>$

Dann $<$Trage [i, j, A, αw_j, β] in C ein$>$

Nur diese Prozedur kann Kanten generieren, die zu einem Punkt führen, zu dem bislang keine Kante reichte:

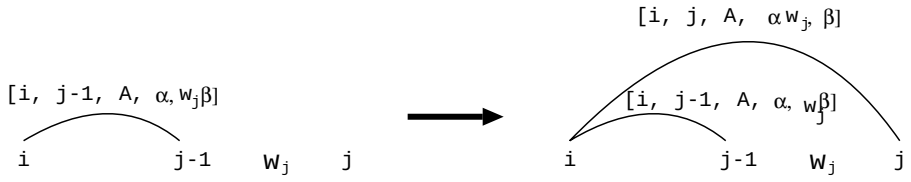


Abbildung (7-4)

Der folgende Erkennungsalgorithmus, der diese drei Prozeduren zur Berechnung der Chart verwendet, generiert zunächst (top-down) alle Kanten, die verschiedene Möglichkeiten repräsentieren, die Analyse des Satzes zu beginnen und erzeugt anschließend von links nach rechts die übrigen Kanten der Chart.

ALGORITHMUS RECOGNIZE_EARLEY-1

DATEN: Ein Lexikon L und eine kontextfreie Syntax G = <V_N, V_T, S, R> ohne Tilgungs- und Kettenregeln.
EINGABE: Ein Satz w = $w_1 \ldots w_n$ (n $\geq$ 1).
AUSGABE: *True*/*False*.
ARBEITSSTRUKTUREN:
Chart - Eine Chart.
Anfangswert: leer.
METHODE:
Initialisierung:
1. Für alle S $\rightarrow \beta \in$ R:
<Trage [0, 0, S, e, β] in Chart ein>
2. Wende EXPAND solange auf die zuvor erzeugten Kanten an, bis keine neuen Kanten mehr erzeugbar sind.

Berechnung der übrigen Kanten:
Für j = 1, ..., n:
Für i = 0, ..., j:
Berechne CHART(i, j):
1. Wende SHIFT auf alle geeigneten Kanten in CHART(i, j-1) an.
2. Wende EXPAND und COMPLETE solange an, bis keine neuen Kanten mehr erzeugt werden können.

Wenn <[0, n, S, β, e] $\in$ CHART>
Dann <RETURN(*True*)>
Sonst <RETURN(*False*)>.

Beispiel (7-3)

Gegeben sei folgende Syntax und folgender Satz:

Syntax: {S $\rightarrow$ (S), S $\rightarrow$ R, R $\rightarrow$ E = E, E $\rightarrow$ (E + E), E$\rightarrow$a, E $\rightarrow$ b}
Satz : (a + b) = b

Die mit dem Algorithmus RECOGNIZE_EARLEY-1 berechnete Chart enthält folgende Kanten:

Chart(0,0) = {S $\rightarrow$.(S), S $\rightarrow$ .R, R $\rightarrow$ .E = E, E $\rightarrow$.(E + E), E $\rightarrow$ .a, E $\rightarrow$ .b}
Chart(0,1) = {*S $\rightarrow$ (.S)*, *E $\rightarrow$ (.E + E)*}
Chart(1,1) = *Chart*(0,0)
Chart(0,2) = {**E $\rightarrow$ (E. + E)**}
Chart(1,2) = {*E $\rightarrow$ a.*, **R $\rightarrow$ E. = E**}
Chart(0,3) = {*E $\rightarrow$ (E +. E)*}
Chart(3,3) = {E $\rightarrow$.(E + E), E $\rightarrow$ .a, E $\rightarrow$ .b}
Chart(0,4) = {**E $\rightarrow$ (E + E.)**}
Chart(3,4) = {*E $\rightarrow$ b.*}

$Chart(0,5) = \{\mathbf{R} \rightarrow \mathbf{E.} = \mathbf{E},\ E \rightarrow (E + E).\}$
$Chart(0,6) = \{R \rightarrow E =.\ E\}$
$Chart(6,6) = Chart(3,3)$
$Chart(0,7) = \{\mathbf{S} \rightarrow \mathbf{R.},\ \mathbf{R} \rightarrow \mathbf{E} = \mathbf{E.}\}$
$Chart(6,7) = \{E \rightarrow b.\}$[5]

7.2.1 Version 1: Berechnung der Chart durch Vorwärtsverkettung

Unbefriedigend bei der Formulierung dieses Algorithmus ist, daß die Interaktion zwischen den drei Operationen nicht genauer spezifiziert wird. Es bietet sich an, in diesem Zusammenhang die zwischen der Prozedur SHIFT und den Prozeduren EXPAND und COMPLETE bestehende Asymmetrie zu nutzen: Nur SHIFT erzeugt Kanten, die den bei der Analyse zuletzt erreichten Punkt mit dem nächsten, bislang unerreichten Punkt verbinden. EXPAND und COMPLETE dagegen generieren ausschließlich solche Kanten, die nur bis zum aktuellen Analysepunkt reichen. Abgesehen von der Initialisierung der Chart setzen sie voraus, daß mindestens eine (durch SHIFT erzeugte) Kante existiert, die bis zum aktuellen Punkt reicht.

Diese Asymmetrie erlaubt es, bei der Berechnung der Chart in jedem Schritt zunächst SHIFT anzuwenden, bevor die Operationen EXPAND und COMPLETE zur Berechnung der übrigen, bis zu diesem Punkt reichenden Kanten (Hüllenbildung) aktiviert werden. Wenn man, wie wir es ja vorausgesetzt haben, davon ausgeht, daß die Syntax keine Tilgungsregeln enthält, dann läßt sich die Hüllenbildung sehr einfach durch *Vorwärtsverkettung* realisieren. Jedesmal wenn eine der beiden Operationen eine neue Kante erzeugt, wird diese neue Kante direkt weiterverarbeitet: ist sie aktiv, wird EXPAND aufgerufen; sonst COMPLETE. Die Hüllenbildung wir durch die Prozedur CLOSURE[6] gesteuert:

PROZEDUR CLOSURE

DATEN: Eine kontextfreie Syntax G, ein Lexikon L und eine Chart C.
EINGABE: Eine Kante k = [i, j, A, α, β].
SEITENEFFEKT: Berechnung von C.

METHODE:
Wenn <k $\notin$ C>
Dann <Trage k in C ein>
  Wenn <k ist eine *passive* Kante>
  Dann <$\underline{COMPLETE_{\mathrm{VK}}}$(k)>
  Sonst <$\underline{EXPAND_{\mathrm{VK}}}$(k)>

Enthält die Syntax keine Tilgungsregeln, werden passive Kanten immer erst dann

[5] Durch SHIFT/COMPLETE erzeugte Einträge sind kursiv/fett gesetzt.
[6] Vgl. [?] und [?].

generiert, nachdem alle aktiven Kanten generiert wurden, mit denen sie kombinierbar sind.
Die Operationen EXPAND und COMPLETE sind so zu modifizieren, daß sie wiederum CLOSURE mit jeder von ihnen generierten Kante aufrufen:

PROZEDUR EXPAND$_{VK}$

DATEN: Eine kontextfreie Syntax G = <V_N, V_T, S, R> und eine Chart C.
EINGABE: Eine Kante k = [i, j, A, α, Bβ].

METHODE:
Für alle B $\rightarrow \tau \in$ R:
<*CLOSURE*([j, j, B, e, τ])>

PROZEDUR COMPLETE$_{VK}$

DATEN: Eine Chart C.
EINGABE: Eine Kante k = [j, l, B, τ, e].

METHODE:
Für alle [i, j, A, α, Bβ] $\in$ C, mit $0 \leq i \leq j \leq l \leq n$:
<*CLOSURE*([i, l, A, αB, β])>

In der vorliegenden Fassung berücksichtigt der Algorithmus noch nicht die von uns geforderte Trennung von Syntax und Lexikon. Das hat zur Folge, daß EXPAND bzw. EXPAND$_{VK}$ für alle [i, j, A, α, Bβ], mit B als einer lexikalischen Kategorie und alle diese Kategorie expandierenden lexikalischen Regeln jeweils eine eigene Kante generiert, ohne zu berücksichtigen, ob das nächste zu verarbeitende Wort des Satzes die Bildung dieser Kante legitimiert oder nicht. Gerade bei linguistischen Anwendungen, in denen es lexikalische Kategorien mit einer großen Zahl von Elementen geben kann, führt dieses Verfahren zu einer indiskutablen Zahl von irrelevanten Kanten.

Die Trennung von Syntax und Lexikon löst dieses Problem: EXPAND$_{VK}$ operiert jetzt nur noch auf syntaktischen Kategorien und eine modifizierte SHIFT-Regel stellt die Generierung geeigneter lexikalischer Kanten sicher:

PROZEDUR SHIFT$_{VK}$

DATEN: Ein Lexikon L.
EINGABE: Eine Wortform w_j des zu analysierenden Satzes, mit $1 \leq j \leq n$.

METHODE:
Für alle lexikalischen Kategorien B, mit $w_j \in$ B:
<*CLOSURE*([j-1, j, B, w_j, e])>

Wir erhalten so als eine Variante des ersten Algorithmus den Algorithmus

RECOGNIZE_EARLEY-2:

ALGORITHMUS RECOGNIZE_EARLEY-2

DATEN: Ein Lexikon L und eine kontextfreie Syntax G = <V_N, V_T, S, R> ohne Tilgungs- und Kettenregeln.
EINGABE: Ein Satz w = $w_1 \ldots w_n$ (n $\geq$ 1).
AUSGABE: *True*/*False*.
ARBEITSSTRUKTUREN:
Chart - Eine Chart.
Anfangswert: leer.
METHODE:
Initialisierung:
Für alle S $\rightarrow \alpha \in$ R:
<*CLOSURE*([0, 0, S, e, α,])>
Berechnung der übrigen Kanten:
Für i = 1, ..., n:
<$SHIFT_{VK}$(w_i)>
Wenn <[0, n, S, α, e] $\in$ CHART>
Dann <RETURN(*True*)>
Sonst <RETURN(*False*)>

Zu beachten bleibt, daß die von $SHIFT_{VK}$ erzeugten Kanten nicht mehr als Erweiterung bereits generierter Kanten, sondern rein bottom-up erzeugt werden. Sie erfüllen aus diesem Grund nicht mehr in jedem Fall die Top-down-Bedingung. $SHIFT_{VK}$ kann aus diesem Grund überflüssige passive Kanten generieren; die Zahl solcher Kanten steht aber in keinem Verhältnis zu der Zahl der anderenfalls durch EXPAND generierten überflüssigen Kanten[7]. Eine Ausnahme bilden nur extreme Formen von lexikalischer Ambiguität.

Wenn lexikalische Kanten bottom-up generiert werden (wie z.B. durch $SHIFT_{VK}$), dann ist es prinzipiell möglich, bei der Berechnung der Chart so zu verfahren, daß alle lexikalischen Kanten in die Chart eingetragen werden, bevor die syntaktischen Kanten berechnet werden. Dieses Vorgehen erlaubt eine schnelle Terminierung des Berechnungsprozesses, wenn der Satz unbekannte Wortformen bzw. Schreibfehler

[7]Eine andere Version von SHIFT erhält man, wenn man diese Prozedur nicht auf terminalen Symbolen, sondern auf lexikalischen Kategorien operieren läßt:

$SHIFT_{VK'}$:
Wenn <[i, j-1, A, α, Bβ] $\in$ C und $w_j \in$ B>
Dann <[i, j, A, αB, β] $\in$ C>

enthält[8]. Allerdings läßt sich die Chart nicht mehr durch einfache Vorwärtsverkettung berechnen, da nun die passiven lexikalischen Kanten vor den aktiven Kanten generiert werden, die durch sie zu ergänzen sind.

Im folgenden Abschnitt entwickeln wir eine Variante des Earley-Algorithmus, bei der die neu generierten Kanten nicht direkt in die Chart eintragen, sondern zunächst zwischengespeichert werden. Sie ermöglicht neben einer Vorberechnung der lexikalischen Kanten auch die Verwendung unterschiedlicher Suchstrategien.

7.2.2 Version 2: Zwischenspeicherung neuer Kanten

In der im letzten Abschnitt vorgestellten Variante des Earley-Algorithmus wird, wenn verschiedene Analysemöglichkeiten zur Auswahl stehen, nach dem Prinzip Depth-first-Suche verfahren. Der wichtigste Unterschied zwischen dieser und der Variante, die wir jetzt entwickeln werden, besteht darin, daß neue Kanten nicht mehr direkt in die Chart eingetragen, sondern zunächst zwischengespeichert werden. Abhängig davon, nach welchem Verfahren die gespeicherten Kanten verwaltet werden, lassen sich völlig unterschiedliche Suchstrategien realisieren; und auch ein durch das Eintreten bestimmter Ereignisse gesteuerter Wechsel der Suchstrategie während des Parsens ist möglich (Agenda-Kontrollstrukturen).

Zur Zwischenspeicherung der neuen Kanten wird eine Liste KANTEN (in [**?**] als „pending edges“ bezeichnet) verwendet. Jede Suchstrategie ist mit einer bestimmten Form des Zugriffs und der Speicherung von Kanten in KANTEN verbunden:

- Wenn KANTEN wie ein Stack verwaltet wird, arbeitet der Algorithmus depth-first.
- Wenn KANTEN wie eine Queue behandelt wird, arbeitet der Algorithmus breadth-first.
- Wenn KANTEN nach 'Qualität' der Kanten sortiert wird (d.h. die für die weitere Analyse des Satzes aussichtsreichsten Kanten vor den weniger vielversprechenden Kanten stehen), dann operiert der Algorithmus nach dem Prinzip der Best-first-Suche.

Unabhängig von der gewählten Suchstrategie kann es geschehen, daß eine passive Kante vor den aktiven Kanten in die Chart eingetragen wird, mit denen sie zu neuen Kanten kombiniert werden kann. Um sicherzustellen, daß trotzdem alle möglichen Kanten gebildet werden, ist es erforderlich, jedesmal wenn eine (aktive/passive) Kante in die Chart eingetragen wird, sie mit allen geeigneten (passiven/aktiven) Kanten der Chart zu kombinieren.

[8] Vgl. [**?**, S.116-26].

ALGORITHMUS RECOGNIZE_EARLEY-3

DATEN: Ein Lexikon L und eine kontextfreie Syntax G = <V_N, V_T, S, R> ohne Tilgungs- und Kettenregeln.
EINGABE: Ein Satz w = $w_1 \ldots w_n$ (n $\geq$ 1).
AUSGABE: *True*/*False*.
ARBEITSSTRUKTUREN:

CHART	-	Eine Chart. Anfangswert: leer.
KANTEN	-	Eine Folge von Kanten. Anfangswert: leer.
K	-	Eine Kante.

METHODE:

Initialisierung:

Chart $\Leftarrow$ {[i-1, i, B, w_i, e] | $w_i \in B$, mit $1 \leq i \leq n$}.
KANTEN $\Leftarrow$ {[0, 0, S, e, α] | S $\rightarrow \alpha \in R$}.

Berechnung der übrigen Kanten:

Wiederhole:
  Wenn <[0, n, S, α, e] $\in$ CHART>
  Dann <RETURN(*True*)>.
  Wenn <KANTEN = $\emptyset$ >
  Dann <RETURN(*False*)>.
  K $\Leftarrow$ *First*(KANTEN).
  KANTEN $\Leftarrow$ *Rest*(KANTEN).
  Chart $\Leftarrow$ Chart $\cup$ {K}.
  KANTEN $\Leftarrow$ KANTEN $\cup$ $\underline{COMPLETE_{PE}}$(K).
  Wenn <K ist eine aktive Kante>
  Dann <KANTEN $\Leftarrow$ KANTEN $\cup$ $\underline{EXPAND_{PE}}$(K)>

Die lexikalischen Kanten werden bei der Initialisierung direkt in die Chart eingetragen; d.h. eine separate Shift-Prozedur ist überflüssig. KANTEN wird mit der Menge aller S-Kanten initialisiert.

Die beiden Operationen $EXPAND_{PE}$ und $COMPLETE_{PE}$ stellen sicher, daß keine Kanten generiert werden, die bereits in CHART oder KANTEN enthalten sind:

PROZEDUR EXPAND$_{\mathrm{PE}}$

DATEN: Eine kontextfreie Syntax G = <V_N, V_T, S, R>, Chart und KANTEN.
EINGABE: Eine Kante k = [i, j, A, α, Bβ].
AUSGABE: Die Menge aller Kanten, durch die k fortgesetzt werden kann.
METHODE:
RETURN({[j, j, B, e, τ] | B $\rightarrow \tau \in$ R $\wedge$ [j, j, B, e, τ] $\notin$ (Chart $\cup$ KANTEN)})

PROZEDUR COMPLETE$_{\mathrm{PE}}$

DATEN: Chart und KANTEN.
EINGABE: Eine Kante k = [i, j, A, α, β].
AUSGABE: Die Menge aller neuen Kanten, die sich durch Kombination von k mit einer Kante aus Chart bilden lassen.
METHODE:
Wenn $< \beta = e >$
Dann <RETURN({ x | x = [h, j, B, σA, τ] $\wedge$ [h, i, B, σ, Aτ] $\in$ Chart $\wedge$ x $\notin$ (Chart $\cup$ KANTEN)})>
Sonst <RETURN({ x | x = [i, l, A, α', β'] $\wedge$ *First*(β) = B $\wedge$ α' = αB $\wedge$ β' = *Rest*(β) $\wedge$ [j, l, B, τ, e] $\in$ Chart $\wedge$ x $\notin$ (Chart $\cup$ KANTEN)}>

7.3 Erweiterung des Algorithmus

7.3.1 Tilgungs- und Kettenregeln

Bislang haben wir vorausgesetzt, daß die Syntax keine Tilgungsregeln enthält. Verwendet man die zuletzt vorgestellte Variante des Algorithmus (RECOGNIZE_EARLEY-3), ist ohne weitere Änderungen eine *direkte* Verarbeitung von Tilgungsregeln möglich. Bei Anwendung des Algorithmus RECOGNIZE_EARLEY-2 können Tilgungsregeln dazu führen, daß die Chart nicht alle prinzipiell generierbaren Kanten enthält: Die Operation COMPLETE$_{\mathrm{VK}}$ generiert neue Kanten, indem sie aktive Kanten durch geeignete passive Kanten ergänzt. Das zuvor beschriebene Verfahren zur Hüllenbildung basiert auf der Voraussetzung, daß die aktiven Einträge immer vor den passenden passiven Einträgen erzeugt werden. Wenn die Syntax Tilgungsregeln enthält, ist diese Voraussetzung nicht mehr generell erfüllt.

Beispiel (7-4)
Syntax : $\{S \rightarrow e, S \rightarrow SAB\}$
Lexikon : $a \in A, b \in B$
Satz : ab
Wenn die Regeln in der oben genannten Reihenfolge verarbeitet werden, enthält die durch RECOGNIZE_EARLEY-2 berechnete Chart nur die beiden Kanten [0, 0, S, e, e] und [0, 0, S, e, SAB], nicht aber die Kante [0, 0, S, S, AB]; d.h. der Satz wird nicht erkannt, obwohl offensichtlich gilt: $S \stackrel{*}{\Rightarrow} ab$.

Es gibt verschiedene Möglichkeiten diesen Algorithmus so zu modifizieren, daß die korrekte Verarbeitung von Tilgungsregeln sichergestellt ist:

1. Bei der Berechnung aller Kanten, die bis zur Position i reichen, werden die Operationen EXPAND und COMPLETE solange angewendet, bis keine neuen Kanten mehr generiert werden[9]. Mit dieser Lösung wird das vergleichsweise elegante und effiziente Verfahren zur Berechnung der Chart durch Vorwärtsverkettung aufgegeben und RECOGNIZE_EARLEY-2 auf die ursprüngliche Fassung des Algorithmus reduziert.

2. Für die zu verarbeitende Syntax G wird vorab die Menge aller tilgbaren Symbole berechnet, und diese Informationen werden bei der Bildung der Chart berücksichtigt[10]:

   Wenn $G = <V_N, V_T, S, R>$ eine beliebige kontexfreie Syntax ist, dann ist für jedes $X \in V_N$ entscheidbar, ob $X \stackrel{*}{\Rightarrow} e$ oder nicht. E_G sei die Menge aller tilgbaren Symbole von G; d.h. $E_G = \{Y \mid Y \in V_N \wedge Y \stackrel{*}{\Rightarrow} e\}$.
   Jedesmal wenn in Chart eine Kante [i, j, A, α, Bβ] eingetragen wird, überprüft man auch, ob $B \in E_G$. Ist das der Fall, wird außerdem die Kante [i, j, A, αB, β] eingetragen und diese neue Kante selbst wieder überprüft (rekursives Testen).

Beispiel (7-5)
Wenn [i, j, X, α ,ABCD] $\in$ Chart und $A, B, C \in E_G$, dann werden auch die Kanten [i, j, X, αA, BCD], [i, j, X, αAB, CD] und [i, j, X, αABC, D] in Chart eingetragen. Diese Lösung läßt sich sehr einfach durch eine Modifikation von CLOSURE realisieren. Ein Nachteil dieser *indirekten* Verarbeitung von Tilgungsregeln besteht darin, daß sie den Aufbau korrekter Strukturbeschreibungen erschwert.

Wenn $X \in E_G$, dann gilt entweder

(a) $X \rightarrow e \in R$ oder

(b) es gibt eine Folge $<\alpha_1, \ldots, \alpha_{n,}>$, mit $n > 1$ und $\alpha_i \in V^*$, so daß gilt:
$\alpha_1 = X$, $\alpha_n \rightarrow e \in R$ und $\alpha_{i-1} \Rightarrow \alpha_i$, mit $1 < i \leq n$.

[9] Vgl. [?, S.321] und [?, S.75].
[10] Vgl. [?, S. 421 ff.].

Wenn es für einen Satz eine Ableitung gibt, in der das Symbol X getilgt wird, dann sollte die dieser Ableitung zugeordnete Strukturbeschreibung alle Schritte abbilden, die zur Tilgung von X führen. Die dazu benötigten Informationen sind aber, wenn Fall (b) vorliegt, nicht verfügbar, sofern sie nicht bei der Berechnung von E_G gespeichert oder separat berechnet werden.

Für eine effiziente Verarbeitung von Kettenregeln gibt es ein ähnliches Verfahren mit entsprechenden Vor- und Nachteilen:

- Für jedes $X \in V_N$ wird die Menge aller Symbole Y_k berechnet, für die gilt: $Y_k \stackrel{*}{\Rightarrow} X$.
- Dann wird die Menge $K_G = \{<\{X, \{Y_1, \ldots, Y_n\}\}> \mid X, Y_i \in V_N \wedge Y_i \stackrel{*}{\Rightarrow} X\ (1 \leq i \leq n)\}$ berechnet.
- Wenn [j, l, X, τ, e] $\in$ CHART, dann wird für jede Kante [i, j, A, α, Bβ] $\in$ CHART, mit B $\in K_G(X)$, auch die Kante [i, l, A, αB, β] in Chart eingetragen.

7.3.2 Look-ahead

Charakteristisch für die in diesem Kapitel entwickelten Algorithmen ist, daß bei der Berechnung der Chart die Generierung der Kanten, die bis zu Position j reichen, allein determiniert wird durch:

1. die bei der Analyse des Satzabschnitts $w_1 \ldots w_{j-1}$ generierten Kanten und
2. den w_j durch das Lexikon zugeordneten lexikalischen Kategorien.

Dieses Verfahren schließt nicht aus, daß (top-down) *überflüssige* Kanten generiert werden, die für die weitere Analyse des Satzes nicht benötigt werden. Die Zahl solcher überflüssigen Kanten läßt sich erheblich verringern, wenn bei der Bildung der Kanten die nächsten k ($k \geq 1$) Worte des noch nicht analysierten Satzabschnitts berücksichtigt werden;[11] d.h. nur solche Kanten generiert werden, die mit den folgenden k Worten 'kompatibel' sind. Grundsätzlich könnte der Look-ahead auf den ganzen Satz bzw. den noch nicht analysierten Teil des Satzes ausgedehnt werden, so daß nur (aktive wie passive) Kanten eingetragen würden, die in einer vollständigen Ableitung des Satzes verwendet werden könnten; d.h. [i, j, A, α, β] $\in$ Chart gdw.

(i). $S \stackrel{*}{\Rightarrow} w_1 \ldots w_i A\tau$;

(ii). $\alpha \stackrel{*}{\Rightarrow} w_{i+1} \ldots w_j$ und

(iii). $\beta\tau \stackrel{*}{\Rightarrow} w_{j+1} \ldots w_n$

Allerdings erfordert die Berücksichtigung des kompletten Rechtskontextes einen erheblichen Rechen- und Speicheraufwand[12], so daß in den meisten Fällen nur mit

[11] Dieses Verfahren, die Zahl der Analysemöglichkeiten durch Betrachtung der nächsten Worte einzuschränken, wird als Parsen mit „Look-ahead“ (*Parsen mit Vorausschauzeichen*) bezeichnet.

[12] Vgl. [?, S.98-100].

einem Look-ahead der Länge 1 gearbeitet wird. Wir werden uns außerdem auf einen partiellen Look-ahead beschränken, der nur die Generierung aktiver Kanten einschränkt; d.h. [i, j, A, α, β] $\in$ CHART, mit $\beta \neq e$, gdw.

(i). s.o.

(ii). s.o.

(iii).' $\beta \stackrel{*}{\Rightarrow} w_{j+1}\ \sigma$

d.h. es werden nur solche aktiven Kanten in die Chart eingetragen, die mit dem nächsten Wort des Satzes kompatibel sind.

Für eine effiziente Realisierung des Look-aheads ist es sinnvoll, für jedes $X \in V_N$ die Menge FIRST(X) zu berechnen, in der alle lexikalischen Kategorien Y enthalten sind, für die gilt: $X \stackrel{*}{\Rightarrow} Y\alpha$. Das Konstruktionsverfahren stellt außerdem sicher, daß $e \in$ FIRST(X) gdw. $X \stackrel{*}{\Rightarrow} e$.

Definition (7-1) *FIRST-Relation*
Wenn $G = <V_N, V_T, S, R>$ eine beliebige kontextfreie Syntax ist, dann wird für jedes $X \in V_N$ die Menge FIRST(X) wie folgt berechnet:

1. Wenn X eine lexikalische Kategorie ist, dann ist FIRST(X) = {X}.
2. Wenn $X \rightarrow e \in R$, dann ist $e \in$ FIRST(X).
3. Wenn $X \in V_N$ und $X \rightarrow Y_1 \ldots Y_n \in R$ $(1 \leq n)$, dann ist $Z \in$ FIRST(X) gdw. $Z \in V_T$, $Z \in$ FIRST(Y_k) $(1 \leq k \leq n)$ und für alle Y_i $(i < k)$ gilt: $e \in$ FIRST(Y_i). Wenn $e \in$ FIRST(Y_1), dann auch $e \in$ FIRST(X).

Die FIRST-Relation läßt sich wie folgt für beliebige $\alpha = X_1 \ldots X_n$ generalisieren:
Wenn $e \notin$ FIRST(X_1)
Dann FIRST(α) = FIRST(X_1)
Sonst FIRST(α) = FIRST(X_1) $\cup$ FIRST($X_2 \ldots X_n$).

Die FIRST-Relation sollte nicht während des Parsens, sondern vorab (Präkompilierung der Syntax) berechnet werden. Der Parsingalgorithmus ist nun so zu modifizieren, daß eine aktive Kante k mit einem offenen Abschnitt α nur dann in Chart(i, j) eingetragen wird, wenn gilt:

$$\text{FIRST}(\alpha) \cap \{ B \mid w_{j+1} \in B \} \neq \emptyset \quad \vee \quad e \in \text{FIRST}(\alpha)^{13}.$$

Die Änderung des Algorithmus bzw. der Prozeduren bleibt dem Leser und der Leserin überlassen (s. Aufgaben am Ende dieses Kapitels).

[13] Zu diesem Zweck kann z.B. ein Prädikat definiert werden, daß alle passiven und die aktiven Kanten erfüllen, die dieser Look-ahead-Bedingung genügen.

7.4 Die Generierung von Strukturbeschreibungen

7.4.1 Explizite Repräsentation

Eine sehr einfache Möglichkeit, Strukturbeschreibungen zu generieren, besteht darin, in den Einträgen der Chart neben einer geteilten Produktion zusätzliche, für ihren Aufbau benötigte Informationen (partielle Strukturbeschreibungen) zu speichern. Nach der Berechnung der Chart werden dann die Strukturbeschreibungen aller passiven S-Kanten ausgegeben, die über den ganzen Satz reichen.

Um aus RECOGNIZE_EARLEY-2 einen nach diesem Prinzip operierenden Parsingalgorithmus zu erhalten[14], ersetzen wir die Operationen SHIFT_{VK}, $\text{EXPAND}_{\text{VK}}$ und $\text{COMPLETE}_{\text{VK}}$ durch:

PROZEDUR $\text{SHIFT}_{\text{VK-P}}$

DATEN: Ein Lexikon L.
EINGABE: Ein Wort des zu analysierenden Satzes w_j, mit $1 \leq j \leq n$.

METHODE:
Für alle lexikalischen Kategorien B, mit $w_j \in B$:
$<\underline{CLOSURE}([j\text{-}1, j, B, w_j, e, <w_j>])>$

PROZEDUR $\text{EXPAND}_{\text{VK-P}}$

DATEN: Eine kontextfreie Syntax $G = <V_N, V_T, S, R>$ und eine Chart C.
EINGABE: Eine Kante $k = [i, j, A, \alpha, B\beta, \sigma]$.

METHODE:
Für alle $B \rightarrow \tau \in R$:
$<\underline{CLOSURE}([j, j, B, e, \tau, < >])>$

PROZEDUR $\text{COMPLETE}_{\text{VK-P}}$

DATEN: Eine Chart C.
EINGABE: Eine Kante $k = [j, l, B, \tau, e, <w_1 \ldots w_n>]$.

METHODE:
Für alle $[i, j, A, \alpha, B\beta, <v_1 \ldots v_m>] \in$ Chart , mit $m = |\alpha|$:
$<\underline{CLOSURE}([i, l, A, \alpha B, \beta, <v_1 \ldots v_m <B <w_1 \ldots w_n>>>])>$

[14] Dieses Verfahren kann natürlich auch bei dem Algorithmus RECOGNIZE_EARLEY-3 verwendet werden.

Ein erheblicher Nachteil dieses Verfahrens besteht darin, daß es relativ speicherintensiv ist: Ähnlich wie es bei einfachen Parsingalgorithmen vorkommen kann, daß Teilstrukturen mehrfach auf dieselbe Art und Weise analysiert werden, wenn bei der Analyse eines Satzes Backtracking ausgelöst wird, kann es hier geschehen, daß die in mehreren Kanten repräsentierten Strukturbeschreibungen partiell identisch sind. Wenn z.B. ein Satz syntaktisch mehrdeutig ist, gibt es pro Lesart in der Chart einen S-Eintrag, der jeweils eine vollständige Strukturbeschreibung enthält.

Beispiel (7-6)

Syntax : {S → NP VP, S → NP VP PP, S → S konj S, NP → det N1,
N1 → adj n, NP → N2 n, N2 → det adj, VP → v adv}

Unter Verwendung der voranstehenden (offensichtlich linguistisch wenig motivierten) Syntax und eines geeigneten Lexikons ergeben sich für den Satz „der alte mann starb heute“ folgende S-Kanten in CHART(0, 5):

[0, 5, S, NP VP, PP,
<<NP <N2 <det der> <adj alte>> <n mann>>
<VP <v starb> <adv heute>>>]
[0, 5, S, NP VP, PP,
<<NP <det der> <N1 <adj alte> <n mann>>>
<VP <v starb> <adv heute>>>]
[0, 5, S, S, konj S,
<<S <NP <N2 <det der> <adj alte>> <n mann>>
<VP <v starb> <adv heute>>>>]
[0, 5, S, S, konj S,
<<S <NP <det der> <N1 <adj alte> <n mann>>>
<VP <v starb> <adv heute>>>>]
[0, 5, S, NP VP, *e*,
<<NP <N2 <det der> <adj alte>> <n mann>>
<VP <v starb> <adv heute>>>]
[0, 5, S, NP VP, *e*,
<<NP <det der> <N1 <adj alte> <n mann>>>
<VP <v starb> <adv heute>>>]

Die Redundanz dieses Verfahrens ist offensichtlich; siehe z.B. die nicht-ambige VP, deren Struktur in allen Kanten explizit repräsentiert wird.

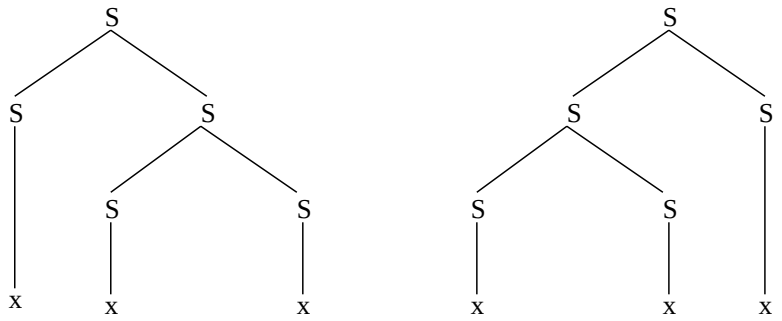
7.4.2 Implizite Repräsentation

Ein sehr viel effizienteres Verfahren entwickeln Graham, Harrison & Ruzzo in [?]. Statt die Strukturen explizit in den Kanten zu repräsentieren, wird in jeder Kante eine Folge von Zeigerpaaren Z = <<A_1, P_1>, <A_2, P_2>, ...<A_n, P_n>> gespeichert[15]. Jedes Zeigerpaar repräsentiert eine Bildungsmöglichkeit für die Kante: dabei weist A_i auf die aktive und P_i auf die passive Kante, aus deren Kombination die Kante resultiert. Voraussetzung für die Realisierung dieses Verfahrens ist eine stabile Indizierung der Kanten der Chart.

Beispiel (7-7)
Betrachtet man die Syntax G = {S → SS, S → x}, dann enthält die Chart für den Satz „xxx“ folgende Kanten:

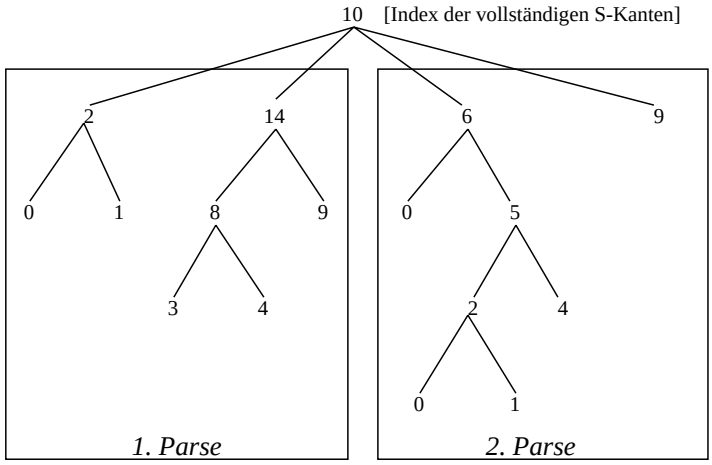
Index	*Kante*
0	[0, 0, S, *e*, SS, < >]
1	[0, 1, S, x *e*, < >]
2	[0, 1, S, S, S, <<0 1>>]
3	[1, 1, S, *e*, SS, < >]
4	[1, 2, S, x, *e*, < >]
5	[0, 2, S, SS, *e*, <<2 4>>]
6	[0, 2, S, S, S, <<0 5>>]
7	[2, 2, S, *e*, SS, < >]
8	[1, 2, S, S, S, <<3 4>>]
9	[2, 3, S, x, *e*, < >]
10	[0, 3, S, SS, *e*, <<2 14> <6 9>>]
11	[0, 3, S, S, S, <<0 10>>]
12	[3, 3, S, *e*, SS, < >]
13	[2, 3, S, S, S, <<7 9>>]
14	[1, 3, S, SS, *e*, <<8 9>>]
15	[1, 3, S, S, S, <<3 14>>]

Die beiden Strukturbeschreibungen des Satzes



[15]Für alle Kanten, die durch COMPLETE gebildet werden, gilt: n ≥ 1; für alle anderen n = 0.

werden implizit durch den aus den Zeigerpaaren der Kanten definierten Baum repräsentiert:



Der Parser muß so konzipiert sein, daß er auf Grundlage derartiger Verweisstrukturen alle Strukturbeschreibungen des Satzes generieren kann. Diese Art der indirekten Repräsentation von Strukturbeschreibungen eignet sich besonders für die Generierung von 'gepackten' Strukturbeschreibungen („local ambiguity packing"), die alle Strukturbeschreibungen eines Satzes zusammenfassen, die einen Abschnitt des Satzes als Konstituente desselben Typs beschreiben, aber jeweils eine andere interne Struktur der Konstituente postulieren.

7.5 Implementierung der Algorithmen

Auf der Basis beider Varianten des Earley-Algorithmus (Berechnung der Chart durch Vorwärtsverkettung / Zwischenspeicherung der neu generierten Kanten) entwickeln wir in diesem Abschnitt einen Recognizer. Für die erste Variante des Algorithmus geben wir eine Prolog-Implementierung an; die zweite wird in Lisp implementiert und anschließend zu einem earley-basierten Parser erweitert, der Strukturbeschreibungen über Verweispaare verwaltet (vgl. 6.4.2) und *gepackte* Strukturbeschreibungen ausgibt.

7.5.1 Prolog

Bei dieser Implementierung des Earley-Algorithmus handelt es sich um eine in Details modifizierte Fassung der Implementierung von Dörre (Vgl. [**?**]). Die Chart wird durch eine Menge von Fakten repräsentiert. Sie haben die Form:

Kante(Start, Ende, Kopf, Geschlossen, Offen).

Start und *Ende* sind Indizes, die den Abschnitt der Eingabekette markieren, auf den sich die Kante bezieht. Für diesen Abschnitt gibt es eine Regel, deren linke Seite

Kopf ist und deren rechte Regelseite aus den Kategorien der Listen *Geschlossen* und *Offen* bestehen. *Geschlossen* ist eine Folge von Kategorien, die zu dem durch *Start* und *Ende* markierten Abschnitt der Eingabekette expandieren, *Offen* ist eine (möglicherweise leere) Liste von Kategorien, die noch keinem Abschnitt der Eingabekette zugeordnet werden konnten.

Repräsentation des linguistischen Wissens

Das Lexikon wird durch Fakten der Form *lex(Wort, Eintrag)* repräsentiert. Um eine der üblichen Repräsentationsform für kontextfreie Regeln möglichst nahe kommende Notation zu ermöglichen, definieren wir einen geeigneten Operator:

```
:- op(1100, xfx, —>).
```

Eine Regel $X \rightarrow Y_1 \ldots Y_n$ ($n \geq 1$) kann jetzt als Prolog-Term X —> [Y_1, ..., Y_n] notiert werden. Die Syntax darf auch Tilgungsregeln enthalten. Sie werden aber nicht wie andere Regeln repräsentiert und direkt verarbeitet, sondern es werden durch Fakten der Form *deleteable(Kategorie)* alle tilgbaren Kategorien markiert.

Hauptprädikate

```
% recognize/1
% Mit diesem Prädikat wird die Analyse eines Satzes gestartet. Der Satz wird durch
% eine Liste von Prolog-Atomen repräsentiert.
recognize(_) :-
    abolish(kante, 5),              % „Aufräumen“ der Datenbasis.
    start_symbol(S),!,
    (S —> X),                       % Initialisierung der Chart.
    closure(0, 0, S, [ ], X), fail.
recognize(Satz) :-
    recognize_recursive(0, Satz).   % Aufruf des Hauptprädikats.

% recognize_recursive/2
recognize_recursive(Start, [Wort | Rest]) :-
    Ende is Start + 1,              % Inkrementierung des Index.
    shift(Start, Ende, Wort),       % Aufruf der Shift-Prozedur.
    recognize_recursive(Ende, Rest). % Rekursion mit Restsatz.
recognize_recursive(Ende, [ ]) :-   % Satz ist vollständig abgearbeitet.
    auswerten(Ende).

% shift/3
% Einlesen der Lexikoneinträge für Wort und Aufruf von CLOSURE für jeden ge-
% fundenen Eintrag.
shift(Start, Ende, Wort) :-
    lex(Wort, Kategorie),           % Lexikonzugriff.
```

```
    closure(Start, Ende, Kategorie, [Wort], [ ]),
    fail.                          % Backtracking: Suche weitere
                                   % Lexikoneinträge für das Wort.
shift(_, _, _).                    % Terminierung.

% closure/5
% Hüllenbildung durch Vorwärtsverkettung.
% Offen ist leer, d.h. es wird eine passive Kante verarbeitet.
closure(Start, Ende, Kopf, Geschlossen, [ ]) :-
    neue_kante(Start, Ende, Kopf, Geschlossen, [ ]),
    complete(Start, Ende, Kopf).
% Die erste Kategorie der Offen-Liste kann getilgt werden und wird deshalb
% als abgearbeitet aufgefaßt; rekursiver Aufruf von closure mit neuer Kante.
closure(Start, Ende, Kopf, Geschlossen, [K | Ks]) :-
    deleteable(K),
    append(Geschlossen, [K], Geschlossen1),
    closure(Start, Ende, Kopf, Geschlossen1, Ks).
% Die erste Kategorie der Offen-Liste kann nicht getilgt werden:
% Top-down-Suche nach Expansionen der ersten Kategorie K in Offen.
closure(Start, Ende, Kopf, Geschlossen, [K | Ks]) :-
    neue_kante(Start, Ende, Kopf, Geschlossen, [K | Ks]),
    expand(Ende, K).

% neue_kante/5
% Es wird geprüft, ob die Kante bereits in der Chart ist. Wenn nicht, wird sie als
% neuer Charteintrag in die Datenbasis geschrieben.
neue_kante(Start, Ende, Kopf, Geschlossen, Offen) :-
    kante(Start, Ende, Kopf, Geschlossen, Offen), !, fail.
neue_kante(Start, Ende, Kopf, Geschlossen, Offen) :-
    assertz(kante(Start, Ende, Kopf, Geschlossen, Offen)).

% complete/3
% Dieses Prädikat wird von closure/5 aufgerufen, wenn gezeigt werden konnte, daß
% der durch Start und Ende markierte Abschnitt unter Kategorie B fällt.
% Wenn es in der Chart Kanten gibt, bei denen die Kategorie B das erste Element
% der Liste Offen ist und der durch diese Kante beschriebene Abschnitt des Satzes
% bis zu der Position reicht, an dem der von B dominierte Abschnitt beginnt, wird
% B in die Liste Geschlossen übertragen und erneut closure aufgerufen; complete
% ergänzt die Chart also bottom-up.
complete(Start, Ende, B) :-
    % Es gibt eine Kante, die bis Start reicht und B als erste Kategorie
```

```
    % in Offen enthält.
    kante(Start1, Start, Kopf, Geschlossen1, [B | Offen]),
    % B wird aus Offen entfernt und in Geschlossen eingefügt.
    append(Geschlossen1, [B], Geschlossen),
    closure(Start1, Ende, Kopf, Geschlossen, Offen),
    fail.                          % Backtracking
complete(_, _, _).                 % Terminierung.

% expand/2
% Die erste Kategorie K in Offen wird zur Bildung neuer Kanten verwendet.
expand(Punkt, K) :-
    (K —> X),                      % Suche Regel, die K expandiert.
    closure(Punkt, Punkt, K, [ ], X),
    fail.                          % Backtracking: Suche weitere Regeln.
expand(_, _).                      % Terminierung
```

Hilfsprädikate

```
% auswerten/1
% Gibt es eine passive Kante für die komplette Eingabekette, die mit dem Start-
% symbol etikettiert ist?
auswerten(Ende) :-
   start_symbol(S),                % Suche Startsymbol.
   kante(0, Ende, S, X, [ ]).      % Suche komplette S-Kante
```

7.5.2 Lisp

Die den folgenden Lisp-Programmen zugrundeliegende Variante des Algorithmus (RECOGNIZE_EARLEY-3) erlaubt es, bei der Berechnung der Chart verschiedene Suchstrategien zu verwenden. Der Parser, den wir am Ende dieses Abschnitts entwickeln werden, verwendet zur Verwaltung der Strukturbeschreibungen eines Satzes eine Technik, die als „*local ambiguity packing*“ bezeichnet wird.

Es wird vorausgesetzt, daß die zur Analyse erforderlichen Daten (Syntax und Lexikon) den beiden globalen Variablen *syntax* und *lexikon* zugewiesen werden. Es ist möglich, nicht nur vollständige Sätze, sondern auch einzelne Konstituenten zu analysieren. In diesem Fall ist beim Funktionsaufruf als zweites Argument eine Liste mit der gesuchten Zielkonstituente anzugeben.

A. Der Recognizer

Selektoren, Konstruktoren & Prädikate

Kantenselektoren

Eine Kante wird durch eine Liste mit 5 Elementen repräsentiert:

(*START ENDE KOPF GESCHLOSSEN OFFEN*)

Die folgenden Funktionen extrahieren jeweils ein Element einer Kante:

```
(defun start (kante)
  (first kante))

(defun ende (kante)
  (second kante))

(defun kopf (kante)
  (third kante))

(defun geschlossen (kante)
  (fourth kante))

(defun offen (kante)
  (fifth kante))
```

Kantenprädikate

```
;;; AKTIVE-KANTE-P
;;; Das Prädikat testet, ob KANTE eine aktive Kante ist.
(defun aktive-kante-p (kante)
  (offen kante))

;;; PASSIVE-KANTE-P
;;; Das Prädikat testet, ob KANTE eine passive Kante ist.
(defun passive-kante-p (kante)
  (endp (offen kante)))

;;; MATCHING-KANTEN-P
;;; Das Prädikat testet, ob KANTE1 mit KANTE2 zu einer neuen Kante verbunden
;;; werden kann.
(defun matching-kanten-p (kante1 kante2)
  (and (eql (ende kante1) (start kante2))
       (eq (first (offen kante1)) (kopf kante2))))
```

Kantenkonstruktor

```
;;; BILDE-KANTE
;;; Diese Funktion bildet aus den beiden Kanten KANTE1 und KANTE2 eine neue
;;; Kante; d.h. KANTE1 wird durch KANTE2 ergänzt.
;;; Vorbedingung: (Matching-Kanten-P KANTE1 KANTE2) ⇒ T.
(defun bilde-kante (kante1 kante2)
  (list (start kante1) (ende kante2) (kopf kante1)
        (append (geschlossen kante1) (list (kopf kante2)))
        (rest (offen kante1))))
```

Hauptfunktion

Bei allen außer dem ersten Parameter der Funktion RECOGNIZE handelt es sich um *keyword*-Parameter, durch die die bei der Analyse zu verwendende Syntax, das Lexikon, wie der Konstituententyp spezifiziert werden können. Der erste Parameter wird an den zu analysierenden Satz (Liste von Symbolen) gebunden. Die drei lokalen Variablen CHART, KANTEN und KANTE werden verwendet, um die Chart, die Menge der neu generierten Kanten und die gerade zu verarbeitende Kante zu verwalten. Die Funktion terminiert, wenn KANTEN leer ist.

```
;;; RECOGNIZE
(defun recognize (satz &key (symbol (startsymbol)))[16]
  (do* ((chart (initialisiere-chart satz))
        (kanten (initialisiere_kanten symbol))
        (kante (first kanten) (first kanten)))
       ((endp kanten) (auswerten chart symbol satz))
  (setf chart (update-chart chart kante)
        kanten (kombiniere (rest kanten) chart
                           (complete chart kante)
                           (when (aktive-kante-p kante) (expand kante))))))
```

Initialisierung von Chart und KANTEN

```
;;; INITIALISIERE-CHART
;;; Diese Funktion bildet auf Grundlage von LEXIKON alle lexikalischen Kanten für
;;; die Worte von SATZ.

(defun initialisiere-chart (satz)
  (let ((resultat nil) (index 0))
    (dolist (wort satz resultat)
      (setf resultat (append resultat (lexikalische-kanten index wort))
            index (+ index 1)))))


;;; LEXIKALISCHE-KANTEN
;;; Diese Funktion generiert für WORT alle möglichen lexikalischen Kanten; d.h. eine
;;; für jede der ihm zugeordneten lexikalischen Kategorien. Jede dieser Kanten reicht
;;; von START bis (+ START 1).

(defun lexikalische-kanten (start wort)
  (mapcar #'(lambda (x) (list start (+ start 1) (first x) nil nil))
          (eintrag-kategorien wort)))
```

[16] Die Definition der Funktionen STARTSYMBOL, REGELN, MATCHING-REGELN und EINTRAG-KATEGORIEN findet sich im Anhang D.1.

```
;;; INITIALISIERE_KANTEN
;;; Diese Funktion generiert für jede Regel aus SYNTAX, deren linke Seite mit
;;; SYMBOL identisch ist, eine aktive Kante mit Start- und Endposition 0.
(defun initialisiere_kanten (symbol)
  (let ((resultat nil))
    (dolist (regel (regeln) resultat)
      (when (eq (first regel) symbol)
        (push (list 0 0 (first regel) nil (rest regel)) resultat)))))
```

Generierung neuer Kanten

```
;;; EXPAND
;;; Diese Funktion nimmt eine aktive Kante KANTE als Argument und generiert für
;;; jede Regel aus SYNTAX, deren linke Seite mit dem ersten Symbol des offenen
;;; Abschnitts von KANTE identisch ist, eine neue aktive Kante, deren Start- und
;;; Endposition mit der Endposition von KANTE identisch ist.
(defun expand (kante)
  (mapcar #'(lambda (x) (list (ende kante) (ende kante) (first (offen kante)) nil x))
          (expansionen (first (offen kante)))))

;;; COMPLETE
;;; Diese Funktion nimmt als Argument eine (aktive/passive) Kante KANTE und
;;; liefert als Wert die Liste aller Kanten, die sich durch Verbindung von KANTE
;;; und einer (passiven/aktiven) Kante aus CHART bilden lassen.
(defun complete (chart kante)
  (if (passive-kante-p kante)
    (mapcar #'(lambda (x) (bilde-kante x kante))
            (remove-if
              #'(lambda (x) (or (passive-kante-p x)
                  (not (matching-kanten-p x kante)))) chart))
    (mapcar #'(lambda (x) (bilde-kante kante x))
            (remove-if
              #'(lambda (x) (or (aktive-kante-p x)
                  (not (matching-kanten-p kante x)))) chart))))
```

Aktualisierung von CHART und KANTEN

Die zur Aktualisierung von CHART und KANTEN notwendigen Funktionen UPDATE und KOMBINIERE sind sehr einfach: UPDATE fügt die aktuelle Kante in die Chart ein. KOMBINIERE ergänzt KANTEN mit allen durch COMPLETE und EXPAND gebildeten Kanten, die nicht in CHART bzw. KANTEN enthalten sind.

```
;;; UPDATE-CHART
;;; Diese Funktion trägt eine neue Kante in die Chart ein.
```

```
(defun update-chart (chart kante)
  (append chart (list kante)))

;;; KOMBINIERE
;;; Diese Funktion kombiniert KANTEN mit den Kantenmengen NEUE-KANTEN1
;;; und NEUE-KANTEN2. Durch Modifikation dieser Funktion können verschiedene
;;; Suchstrategien realisiert werden. Aktuelle Suchstrategie: depth-first.

(defun kombiniere (kanten chart neue-kanten1 neue-kanten2)
  (let ((alte-kanten (append kanten chart)))
    (append kanten
            (remove-if #'(lambda (x) (member x alte-kanten :test #'equal))
                       (append neue-kanten1 neue-kanten2)))))
```

Auswertung von CHART

```
;;; AUSWERTEN
;;; Diese Funktion evaluiert zu T gdw. CHART eine SYMBOL-Kante enthält, die
;;; sich über den ganzen Satz erstreckt; sonst zu NIL.

(defun auswerten (chart symbol satz)
  (dolist (kante chart)
    (when (and (passive-kante-p kante)
               (eq (kopf kante) symbol)
               (eq (start kante) 0)
               (eq (ende kante) (length satz)))
      (return t))))
```

B. Der Parser

Das folgende Programm unterscheidet sich von dem letzten Programm vor allem in zwei Punkten:

1. Es wird eine andere Datenstruktur zur Repräsentation von Kanten verwendet. Kanten werden durch zweielementige Listen der folgenden Form repräsentiert:

   ((*START ENDE KOPF GESCHLOSSEN OFFEN*) ($P_1 \ldots P_n$))
   KERN POINTER

   Das erste Element wird als „Kantenkern“ bezeichnet; das zweite Element ist eine Liste, die die zur Generierung der Strukturbeschreibung notwendigen Verweise (Pointer) enthält.

2. Neu hinzugekommen sind eine Reihe von Funktionen, die zur Generierung der Strukturbeschreibungen benötigt werden.

Außer den neuen Funktionen beschreiben wir in diesem Abschnitt nur diejenigen Funktionen, die gegenüber dem vorangegangenen Progamm erheblich modifiziert wurden.

Selektoren, Konstruktoren & Prädikate

Kantenselektoren

Aufgrund der geänderten Repräsentation von Kanten sind folgende Kantenselektoren zu ändern:

START	GESCHLOSSEN	KOPF
ENDE	OFFEN	

Neu hinzu kommen zwei Selektoren, die den Kantenkern bzw. die Pointerliste einer Kante als Wert liefern:

```
(defun kern (kante)
  (first kante))
```

```
(defun pointer (kante)
  (second kante))
```

Kantenkonstruktor

Anders als beim Recognizer wird bei BILDE-KANTE als drittes Argument die Chart mit übergeben, da für die Bildung der neuen Kante die Indizes von KANTE1 und KANTE2 berechnet werden müssen. Jede durch COMPLETE/BILDE-KANTE generierte Kante enthält in POINTER die Indizes der beiden Kanten, die ihre Generierung rechtfertigen: Der erste Index ist der Index der aktiven, der zweite der Index der passiven Kante.

```
;;; BILDE-KANTE
;;; Generiert eine neue (aktive/passive) Kante.
(defun bilde-kante (kante1 kante2 chart)
  (list (list (start kante1)
              (ende kante2)
              (kopf kante1)
              (append (geschlossen kante1) (list (kopf kante2)))
              (rest (offen kante1)))
              (list (list (index kante1 chart) (list (index kante2 chart))))))

;;; INDEX
;;; Berechnet den Index von KANTE; d.h. die Position von KANTE in CHART.
;;; Mehrfachvorkommen einer Kante sind aufgrund des Konstruktionsverfahrens für
;;; CHART ausgeschlossen. Index der ersten Kante der Chart: 0.
(defun index (kante chart)
  (let ((found (assoc (kern kante) chart :test #'equal)))
    (if found
```

```
      (- (length chart) (length (member found chart :test #'equal)))
      (break "INDEX : kante not found in Chart"))))
```

Neue Kantenprädikate

```
;;; ZYKLISCHE-KANTE-P
;;; Das Prädikat evaluiert zu T gdw. KANTE eine zyklische Kante ist.
(defun zyklische-kante-p (kante)
  (= (start kante) (ende kante)))

;;; NEUE-KANTE-P
;;; Das Prädikat evaluiert zu T gdw.  es keine Kante in CHART gibt, deren Kern
;;; mit dem von KANTE identisch ist.
(defun neue-kante-p (kante chart)
  (not (assoc (kern kante) chart :test #'equal)))
```

Hauptfunktion

Die Funktion PARSE unterscheidet sich von der Funktion RECOGNIZE dadurch, daß KANTEN nur dann durch die von COMPLETE/EXPAND generierten Kanten ergänzt wird, wenn KANTE eine zyklische Kante (d.h. eine durch EXPAND generierte Kante) oder eine neue Kante im Sinne von NEUE-KANTE-P ist. Gibt es eine Kante in der Chart, deren *Kern* mit dem von KANTE identisch ist, wird nur die Verweisliste der Kante durch die Verweise in POINTER(KANTE) ergänzt.

```
;;; PARSE
;;; Hauptfunktion: Strukturgenerierung durch Pointer.
(defun parse (satz &key (symbol (startsymbol)))
  (do* ((chart (initialisiere-chart satz))
        (kanten (initialisiere_kanten symbol) (rest kanten))
        (kante (first kanten) (first kanten)))
       ((endp kanten) (auswerten chart symbol satz))
    (if (or (zyklische-kante-p kante) (neue-kante-p kante chart))
        (setf chart (update-chart chart kante)
              kanten (kombiniere kanten chart
                                 (complete chart kante)
                                 (when (aktive-kante-p kante) (expand kante syntax))))
        (update-kante kante chart))))
```

Initialisierung von CHART und KANTEN

Die Funktionen LEXIKALISCHE-KANTEN und INITIALISIERE_KANTEN sind der geänderten Repräsentation der Kanten anzupassen.

Generierung neuer Kanten

Die Funktionen EXPAND und COMPLETE sind der geänderten Repräsentation

der Kanten anzupassen.

Aktualisierung von CHART und KANTEN

```
;;; UPDATE-KANTE
;;; Diese Funktion aktualisiert mit dem Verweispaar von KANTE die Verweisliste der
;;; Kante von CHART, deren Kern mit dem von KANTE identisch ist (!destruktive
;;; Funktion!). UPDATE-CHART evaluiert in diesem Fall zu NIL (vgl. BUILD).
(defun update-kante (kante chart)
  (let ((alte-kante (assoc (kern kante) chart :test #'equal)))
    (setf (pointer alte-kante)
          (add (first (pointer kante)) (pointer alte-kante)))))

;;; ADD
;;; ADD ergänzt die Verweisliste einer Kante durch ein weiteres Verweispaar.
(defun add (verweise verweisliste)
  (let ((x (assoc (first verweise) verweisliste)))
    (if x
        (if (member (first (second verweise)) (second x))
            verweisliste
            (cons (list (first verweise) (append (second verweise) (second x)))
                  (remove x verweisliste :test #'equal)))
        (cons verweise verweisliste))))

;;; KOMBINIERE
;;; Diese Funktion kombiniert KANTEN mit den Kantenmengen NEUE-KANTEN1
;;; und NEUE-KANTEN2. Durch Modifikation dieser Funktion lassen sich verschie-
;;; dene Suchstrategien realisieren.
;;; Aktuelle Suchstrategie: depth-first.
(defun kombiniere (kanten chart neue-kanten1 neue-kanten2)
  (let ((alte-kanten (append kanten chart)))
    (append kanten
            neue-kanten1
            (remove-if #'(lambda (x) (member x alte-kanten :test #'equal))
                       neue-kanten2))))
```

Strukturgenerierung

Die Generierung der *gepackten* Strukturbeschreibungen ist relativ aufwendig, wenn auch immer noch erheblich effizienter als die sequentielle Generierung aller einfachen Strukturbeschreibungen. Jede Kante - mit Ausnahme der durch EXPAND generierten *zyklischen* Kanten (POINTER(k) = NIL) und der durch LEXIKALISCHE-KANTEN erzeugten *lexikalischen* Kanten (POINTER(k) = (*Wort*)) - wird durch Verknüpfung einer *aktiven* und einer *passiven* Kante gebildet.

Für jede Kante k könnte POINTER(k) einfach aus einer Liste von Verweispaaren $((A_1\ P_1) \ldots (A_n\ P_n))$ $(n \geq 1)$ bestehen, mit A_i/P_i als Verweis auf eine aktive und eine passive Kante, deren Kombination k ergibt. Die Länge der Liste, so liegt es nahe zu vermuten, ließe sich als Maß der Ambiguität der durch k repräsentierten Konstituente auffassen. Dieser Ansatz ist allerdings, wie sich schnell zeigen läßt, inadäquat. Wenn k eine beliebige (nicht-zyklische und nicht-lexikalische) Kante ist, dann müssen zwei Möglichkeiten unterschieden werden:

1. Es gibt verschiedene passive Kanten $k_{p1}, \ldots, k_{pm}$ $(m > 1)$, die mit **einer** aktiven Kante k_a zu k verbunden werden können; d.h. es liegt keine direkte lokale Ambiguität vor: Nicht die durch k, sondern die durch $k_{p1}, \ldots, k_{pm}$ repräsentierte Konstituente ist lokal ambig.

2. Es gibt verschiedene Verweispaare $<k_a, k_p>$, die sich durch die Segmentierung des der Kante k zugeordneten Satzabschnitts unterscheiden; erst in diesem Fall liegt eine direkte lokale Ambiguität der durch k repräsentierten Konstituente vor.

Wir haben aus diesem Grund für die Verweisliste folgende Struktur gewählt:

$$((A_1\ (P_{11} \ldots P_{1n})) \ldots (A_r\ (P_{r1} \ldots P_{rm}))),$$

mit $P_{i1} \ldots P_{in}$, mit $1 \leq i \leq r$ und $1 \leq n$, m als Verweise auf eine ambige (passive) Konstituente.

```
;;; AUSWERTEN
;;; Diese Funktion generiert alle Strukturbeschreibungen für alle Kanten vom Typ
;;; SYMBOL, die sich über den ganzen Satz erstrecken.
(defun auswerten (chart symbol satz)
  (dolist (kante chart (format t "%%keine weiteren parse gefunden!"))
    (when (and (passive-kante-p kante)
               (eq (kopf kante) symbol)
               (eq (start kante) 0)
               (eq (ende kante) (length satz)))
      ;; lokale Ambiguitäten, die über den ganzen Satz reichen, werden
      ;; nicht gepackt:
      (mapcar #'(lambda (x) (pprint (struktur (list (kern kante) (list x)) chart)))
              (pointer kante)))))

;;; STRUKTUR
;;; Generiert aus KANTE alle Strukturbeschreibungen. Ambige Substrukturen wer-
;;; den 'gepackt' repräsentiert („lokal ambiguity packing"). Gepackte Konstituenten
;;; sind durch doppelte Klammerung und Identität des ersten Symbols der beiden
;;; Teilstrukturen erkennbar.
;;; Beispiel: ... ((NP (NP1 ...) ...) (NP (NP2 ...) ...) ...) ...
```

```
(defun struktur (kante chart) (when (passive-kante-p kante) (list (kopf kante))))
  (cond ((endp (pointer kante))
         ((symbolp (first (pointer kante))) (cons (kopf kante) (pointer kante)))
         (t (verarbeite (pointer kante) kante chart))))

;;; VERARBEITE
;;; Für jeden Eintrag der Verweisliste wird eine Strukturbeschreibung generiert.
(defun verarbeite (verweisliste kante chart)
  (let ((resultat nil))
    (dolist (verweise verweisliste (if (rest resultat) (list resultat) (first resultat)))
      (push (generiere-struktur kante
                (struktur (nth (first verweise) chart) chart)
                (if (rest (second verweise))
                    (mapcar #'(lambda (x) (struktur (nth x chart) chart))
                            (second verweise))
                    (struktur (nth (first (second verweise)) chart) chart)))
            resultat))))

;;; GENERIERE-STRUKTUR
;;; Wenn KANTE ein aktive Kante ist, wird nur die mit KANTE korrespondierende
;;; Struktur zurückgegeben; sonst muß auch der Kopf von KANTE in die Struktur
;;; aufgenommen werden.
(defun generiere-struktur (kante struktur1 struktur2)
  (let ((struktur (kombiniere-strukturen kante struktur1 struktur2)))
    (if (aktive-kante-p kante) struktur
        (if (atom (first struktur)) (list (kopf kante) struktur)
            (cons (kopf kante) struktur)))))

;;; KOMBINIERE-STRUKTUREN
;;; Diese Funktion kombiniert die Strukturen STRUKTUR1 und STRUKTUR2, die
;;; mit zwei Kanten korrespondieren, aus denen KANTE gebildet werden kann.
(defun kombiniere-strukturen (kante struktur1 struktur2)
  (cond ((endp struktur1) struktur2)
        ((and (= (length (geschlossen kante)) (+ (length struktur1) 1))
              (listp (first struktur1)))
         (append struktur1 (list struktur2)))
        (t (list struktur1 struktur2))))
```

Bei der Kombination der beiden Substrukturen STRUKTUR1 und STRUKTUR2 sind drei Fälle zu unterscheiden:

1. STRUKTUR1 ist leer (d.h. KANTE ist eine zyklische Kante); d.h. KOMBINIERE-STRUKTUREN evaluiert zu STRUKTUR2.

2. Beide Teilstrukturen müssen zu einer koordinierten Struktur verbunden werden, d.h., die Top-Level-Elemente von STRUKTUR1 und STRUKTUR2 sind auch die Top-Level-Elemente der neuen Struktur.

3. STRUKTUR1 ist eine 'gepackte' Struktur oder besteht aus einer Liste, die ein Symbol enthält (basierend auf einer Tilgungsregel). Dann sind beide Teilstrukturen zu einer subordinierten Struktur zu verbinden; d.h. die neue Struktur enthält STRUKTUR1 und STRUKTUR2 als Top-Level-Elemente.

Aufgaben

7.1 Schreiben Sie ein Prolog-/Lisp-Programm, das für alle nicht-terminalen Symbole einer kontextfreien Syntax die FIRST-Relation berechnet.

7.2 Modifizieren Sie beide Versionen des Earley-Algorithmus so, daß nur Kanten in die Chart eingetragen werden, die die Look-ahead-Bedingung erfüllen. Implementieren Sie die geänderten Algorithmen in Prolog/Lisp.

7.3 Überlegen Sie, wie der Earley-Algorithmus zu modifizieren ist, damit Regeln verarbeitet werden können, in denen fakultative Symbole vorkommen oder einzelne Symbole beliebig oft auftreten können (vgl. 6.1). Gehen Sie davon aus, daß Symbole nicht mehrfach markiert sein können.

7.4 Bei der Entwicklung von Recognizern ist es nicht sinnvoll, zunächst die Chart vollständig zu berechnen, um anschließend zu überprüfen, ob sie mindestens eine Kante enthält, die den Satz als syntaktisch wohlgeformt qualifiziert. Modifizieren Sie die in diesem Kapitel formulierten Erkennungsalgorithmen so, daß die Berechnung der Chart terminiert, sobald eine derartige Kante generiert wurde, und implementieren Sie sie.

7.5 Wenn es für das nächste zu analysierende Wort keinen Lexikoneintrag gibt, terminieren die in diesem Kapitel beschriebenen Algorithmen mit *false*. Eine andere Möglichkeit besteht darin, die Top-down-Komponente des Algorithmus zu verwenden, um Hypothesen über Kategorisierung des Wortes zu generieren. Entwickeln Sie ein Programm, das solche Hypothesen bildet und sie dazu verwendet, den Satz weiter zu analysieren und das Lexikon zu erweitern.

Kapitel 8

Insel-Parsing

Alle Algorithmen, die wir in den vorangegangenen Kapiteln formuliert haben, verarbeiten Sätze Wort für Wort von links nach rechts. Der Algorithmus, den wir im folgenden beschreiben werden, operiert im Gegensatz zu diesen Algorithmen nicht unidirektional, sondern kann, nachdem ein bzw. mehrere Wort(e) des Satzes als *Inseln* markiert wurden, die Analyse in beide Richtungen fortsetzen[1]. Grundlage des Algorithmus bildet der Earley-Algorithmus (vgl. Kapitel 7).

8.1 Motivation für Insel-Parsing

Entscheidend für die Entwicklung von Insel-Parsingalgorithmen waren die Probleme, die sich bei der maschinellen Verarbeitung gesprochener Sprache ergeben können: Während bei einer textuellen Repräsentation von Sprache die Erkennung der Wort- und Satzgrenzen in der Regel keine Probleme bereitet, da Satz- bzw. Leerzeichen relativ verläßlich Einheitsgrenzen markieren, kann es sich bei der Analyse akustischer Signale als äußerst schwierig erweisen, die für das Parsen relevanten Einheiten eindeutig zu identifizieren. Aus diesem Grund liefert die akustisch-phonetische Analyse von Äußerungen normalerweise nicht eine Folge zweifelsfrei erkannter Wortformen, sondern eine Menge z.T. inkompatibler Worthypothesen, die zu ganz unterschiedlichen Analyseergebnissen führen können.

Beispiel (8-1)

[1] Vgl. [?], [?] und [?].

So könnte die Äußerung

w i: k ɔ: t s i k s i: l z w i ð i n æ d i k w i t b e i t s

zu den folgenden beiden Hypothesen führen, die nur eine gemeinsame Worthypothese (baits) aufweisen:

(a)

week	ought	sick	seals	within	adequate	baits
Woche	*sollte(n)*	*krank*	*Siegel*	*innerhalb*	*geeignet(e)*	*Köder*

(b)

we	caught	six	eels	with	inadequate	baits
Wir	*fingen*	*sechs*	*Aale*	*mit*	*ungeeigneten*	*Ködern*

Eine naheliegende Strategie, die eine effiziente Lösung dieses Problems und eine weitgehende Vermeidung falscher Analysen wie (8-1)(a) ermöglicht, besteht darin, sich beim Parsen zunächst an den Worthypothesen zu orientieren, die die akustisch-phonetische Analyse als besonders verläßlich markiert hat (*Inseln*).

Außerdem sollte versucht werden, bei der weiteren Analyse der Äußerung den durch die Hypothesen determinierten Suchraum durch Rekurs auf verfügbare linguistische Wissensquellen (lexikalisch-morphologisches, syntaktisches und semantisches Wissen) möglichst stark zu reduzieren[2].
Da diese Worthypothesen sich auf ein beliebiges Segment und nicht unbedingt auf das Anfangs- bzw. Endsegment der Äußerung beziehen können, muß der Algorithmus es ermöglichen, die Analyse von den sicher erkannten Wortformen aus in beide Richtungen fortzusetzen.

Aber nicht nur bei der Analyse gesprochener Sprache kann die Verwendung eines Insel-Parsingalgorithmus sich als sinnvoll erweisen: In den meisten neueren Grammatiktheorien und -formalismen (z.B. *Government-&-Binding-Theorie, Lexical-Functional Grammar, Generalized Phrase Structure Grammar* und *Head-Driven Phrase Structure Grammar*) wird die Form zulässiger Regeln durch die $\bar{X}$-Konvention eingeschränkt[3]. Diese Konvention fordert, daß jede Regel auf der rechten Seite eine *Kopfkategorie* („head") enthalten muß. Diese Kopfkategorie determiniert den Typ und die wichtigsten syntaktischen wie semantischen Eigenschaften der durch die Regel beschriebenen Konstituente: So ist z.B. in Regeln der Form $\mathrm{VP} \rightarrow \ldots \mathrm{V} \ldots$ V die Kopfkategorie, und N ist die Kopfkategorie der Regeln der Form $\mathrm{NP} \rightarrow \ldots \mathrm{N} \ldots$.

Ähnlich wie bei der Analyse von Äußerungen die verläßlichen Worthypothesen die Startpunkte der Analyse markieren, können hier die Wortformen als Inseln ausgewählt werden, denen das Lexikon eine solche Kopfkategorie zuordnet[4].

Beispiel (8-2)
Wenn wir N(omen) und V(erb) als Kopfkategorien auszeichnen, könnten die ersten Schritte einer bidirektionalen Analyse des Satzes

„Er verlor den Kopf, als das Dessert auf seinem Sweat-Shirt landete."

in folgender Weise ablaufen:
(1). Identifikation der Inseln:

N	V		N		N		N	V
er	verlor	...	kopf	...	dessert	...	sweat-shirt	landete

(2). Erweiterung der Inseln:

[2]Zu den klassischen wissensbasierten Systemen in diesem Bereich gehören z.B. die ARPA-Systeme (*Advanced Research Projects Agency*) HARPY [?] und HEARSAY II [?].

[3]Eine gut lesbare Darstellung der $\bar{X}$-Konvention findet sich z.B. in: [?, S.27-33].

[4]Ein weiteres Argument für das Insel-Parsing liefern bestimmte Konstruktionen in natürlichen Sprachen, die bei unidirektionaler Analyse Schwierigkeiten bereiten; vgl. z.B. die Behandlung von Koordinationen bei [?, S.227-29].

(a) NP er ... NP *den*[5]Kopf ... NP *das* dessert ... NP *seinem* sweat-shirt ...

(b) ... VP verlor *NP* ..... PP auf *NP* ...

(c) ..... VP *PP* landete

(d) ..... S' *als NP* VP

Die Auszeichnung bestimmter Wörter bzw. Kategorien als Inseln kann auch in diesem Fall, wie wir später an einem Beispiel zeigen werden, die Effizienz der Analyse erheblich beeinflußen.

8.2 Notwendige Modifikationen des Earley-Algorithmus

Bei der Entwicklung eines earley-basierten Insel-Parsingalgorithmus verfügt man über einen gewissen Gestaltungsspielraum. Es gibt mindestens drei Parameter, für die man verschiedene Werte wählen kann:

1. Die Zahl der zulässigen Inseln:
   (a) eine Insel;
   (b) eine beliebige Zahl von Inseln.
2. Die Methode zur Identifikation der Inseln:
   (a) Auszeichnung von lexikalischen Kategorien;
   (b) direkte Markierung eines Wortes bzw. mehrerer Wörter innerhalb des Satzes (z.B. *Zeichne das erste Wort als Insel aus* $\Longrightarrow$ normale links-rechts Analyse.).
3. Die beim Insel-Parsen verwendete Analysestrategie:
   (a) Übernahme der Analysestrategie des Earley-Algorithmus;
   (b) Verwendung einer Analysestrategie, die durch die der Bottom-up-Aspekt stärker betont wird[6].

Der Algorithmus, den wir im folgenden entwickeln werden, zeichnet sich durch folgende Eigenschaften aus:

[5]Ergänzungen der Kopfkategorie einer Konstituente sind in diesem Beispiel durch Kursivschrift hervorgehoben.

[6]Natürlich lassen sich diverse Analysestrategien für das Insel-Parsing verwenden und so z.B. ein Insel-Parsingalgorithmus formulieren, der vollständig bottom-up arbeitet (vgl. [?]) oder der die Left-corner-Strategie verwendet.

- Es werden beliebig viele Inseln pro Satz zugelassen.
- Die Inseln werden durch Auszeichnung von lexikalischen Kategorien determiniert.
- Die Analyse verläuft weitgehend bottom-up: Es wird eine neue, rein bottom-up arbeitende Operation (REDUCE) definiert, und die Anwendung von EXPAND wird eingeschränkt.

8.2.1 Datenstrukturen und Operationen

Wir erweitern den Earley-Algorithmus in zwei Schritten: Zunächst ändern wir die Struktur der Kanten, die wir in die Chart eintragen, und anschließend modifizieren bzw. ergänzen wir die Grundoperationen des Earley-Algorithmus so, daß sie eine bidirektionale Verarbeitung von Sätzen gestatten.

Kanten. Solange wie in den bislang betrachteten Algorithmen Teilstrukturen unidirektional erkannt bzw. gebildet werden, ist es ausreichend, sie so zu repräsentieren, daß nur zwischen dem erkannten und dem noch nicht erkannten Teil der Struktur (*offener / geschlossener Abschnitt* einer Kante) unterschieden wird. Für das Insel-Parsing dagegen sollte eine Repräsentation gewählt werden, die explizit zwischen

(a) dem erkannten Teil T einer Struktur (*geschlossener Abschnitt*),

(b) dem nicht erkannten Teil links von T (*linker Abschnitt*) und

(c) dem nicht erkannten Teil rechts von T (*rechter Abschnitt*)

unterscheidet; d.h., wenn man von weiteren, zur Generierung von Strukturbeschreibungen notwendigen Informationen absieht, haben Kanten nun die Form:

$$\text{Kante} := [\textit{anfang ende kopf links geschlossen rechts}].$$

Natürlich lassen sich Kanten weiterhin wie gewohnt als geteilte Produktionen notieren:

$$\textit{Kopf} \rightarrow \textit{links.geschlossen.rechts} \in \text{Chart}(\textit{anfang}, \textit{ende}).$$

Operationen. Mit einem ähnlich geringen Aufwand lassen sich auch die vom Earley-Algorithmus zur Berechnung der Chart verwendeten Prozeduren (EXPAND und COMPLETE) so modifizieren, daß sie eine bidirektionale Verarbeitung des Satzes erlauben.

Die Prozedur COMPLETE bildet aus zwei Kanten k_1 = [i, j, X, α, Yγ] und k_2 = [j, l, Y, β, e] eine neue Kante k_3 = [i, l, X, αY, γ]. Die für das Insel-Parsing zu formulierende Operation $\text{COMPLETE}_{\text{IS}}$ muß zwischen Links- und Rechtserweiterung unterscheiden:

- *Rechtserweiterung*:
  Aus den Kanten [j, l, Y, e, β, e], [i, j, X, δ, α, Yγ] kann COMPLETE$_{IS}$ die Kante [i, l, X, δ, αY, γ] bilden.

- *Linkserweiterung*:
  Die Kanten [j, l, Y, e, β, e], [l, m, Z, τY, ρ, σ] können zur Kante [j, m, Z, τ, Yρ, σ] kombiniert werden.

Diese Lösung ist allerdings nicht ganz unproblematisch: Wenn die Chart eine aktive Kante enthält, die links und rechts erweitert werden kann, dann kann diese Situation zur Generierung redundanter Kanten führen:[7].

Beispiel (8-3)

Wenn die Chart die Kanten

(1) [i, j, X, A, γ, B]
(2) [h, i, A, e, α, e]
(3) [j, l, B, e, β, e]

enthält, dann werden durch COMPLETE$_{IS}$ die folgenden Kanten generiert werden:

(4) [h, j, X, e, Aγ, B] aus (1) und (2)
(5) [i, l, X, A, γB, e] aus (1) und (3).

Im nächsten Schritt erhalten wir dann:

(6) [h, l, X, e, AγB, e] aus (4) und (3)
(6') [h, l, X, e, AγB, e] aus (5) und (2).

Die Redundanz dieses Verfahrens besteht nicht allein darin, daß dieselbe Kante mehrfach generiert und in die Chart eingetragen wird (letzteres läßt sich durch einen einfachen Test verhindern); auch auf die Generierung der Kante (4) oder der Kante (5) kann verzichtet werden, ohne daß dadurch das Resultat der Analyse beeinflußt wird. Es gibt verschiedene Möglichkeiten zu verhindern, daß durch die Kombination zweier Kanten redundante Kanten generiert und in die Chart eingetragen werden:

1. Man formuliert ein allgemeines Redundanzkriterium, das sicherstellt, daß nur nicht-redundante Kanten generiert werden können (siehe z.B. **r/check** in: [**?**, S.229]). Der Nachteil dieses Verfahrens liegt in der Komplexität des Kriteriums und der Notwendigkeit, alle potentiellen Kantenpaare vor Anwendung der Kantenbildungsoperationen zu testen.

2. Die Operation COMPLETE$_{IS}$ wird so modifiziert, daß Kanten, die nach links und nach rechts erweitert werden können, in einem Schritt beidseitig erweitert werden; d.h. COMPLETE$_{IS}$ kombiniert in diesem Fall drei Kanten (die Kante (6) wird direkt generiert, d.h. ohne daß zuvor die Kanten (4)/(5) gebildet werden).

[7] Vgl. [**?**, S. 228-229].

3. Es wird festgelegt, daß in den Fällen, in denen eine Kante beidseitig erweitert werden kann, sie immer nur in eine Richtung erweitert wird (es wird nur die Kante (4) oder die Kante (5) und damit nur (6) bzw. (6') gebildet).

Wir verwenden wie [**?**] wegen ihrer Einfachheit die letzte Lösung (d.h. wir erweitern beidseitig erweiterbare Kanten immer nur nach rechts) und erhalten so die folgende Version der Operationen $\text{COMPLETE}_{\text{IS}}$ und $\text{EXPAND}_{\text{IS}}$:

Wenn der rechte Abschnitt der Kante nicht leer ist, produziert $\text{EXPAND}_{\text{IS}}$ zyklische Kanten, durch die sie nach rechts erweitert werden kann; sonst zyklische Kanten, die potentielle Kandidaten für eine Linkserweiterung bilden.

PROZEDUR $\text{EXPAND}_{\text{IS}}$

DATEN: Eine kontextfreie Syntax G = <V_N, V_T, S, R> und eine Chart C.
EINGABE: Eine aktive Kante k = [i, j, A, α, β, γ]; d.h. $\alpha \neq e$ oder $\gamma \neq e$.

METHODE:

Wenn $< \gamma = B\gamma' >$
Dann <Für alle $B \rightarrow \tau \in R$:
<Trage [j, j, B, e, e, τ] in C ein>>
Sonst <Für alle $D \rightarrow \delta \in R$, mit $\alpha = \alpha' A$:
<Trage [i, i, D, δ, e, e] in C ein>>

PROZEDUR $\text{COMPLETE}_{\text{IS}}$

DATEN: Eine Chart C.
EINGABE: Eine passive Kante k = [j, l, B, e, τ, e] $\in$ C.

METHODE:

Für alle [i, j, A, α, β, Bγ] $\in$ C, mit $0 \leq i \leq j \leq l \leq n$:
<Trage [i, l, A, α, βB, γ] in C ein>

Für alle [l, m, A, αB, β, e] $\in$ C, mit $0 \leq j \leq l \leq m \leq n$:
<Trage [j, m, A, α, Bβ, e] in C ein>

Die Prozedur $\text{COMPLETE}_{\text{IS}}$ genügt der oben formulierten Restriktion: Kanten, die beidseitig erweiterbar sind, werden immer nur nach rechts erweitert.

Um die Bedeutung der Inseln für den Analyseprozeß zu verstärken, modifizieren wir die Analysestrategie des Earley-Algorithmus so, daß die Anwendung der EXPAND-Prozedur eingeschränkt wird und die meisten Kanten bottom-up generiert werden:

1. Wir initialisieren die Chart nicht mehr durch EXPAND bzw. $\text{EXPAND}_{\text{IS}}$, sondern tragen zunächst nur die lexikalischen Kanten ein, die keine Inseln repräsentieren und benutzen die Inseln zur Steuerung der Analyse.

2. Wir definieren eine weitere, bottom-up arbeitende Operation $\text{REDUCE}_{\text{IS}}$: Ausgehend von einer passiven Kante k generiert $\text{REDUCE}_{\text{IS}}$ für jedes Vorkommen des Kopfes von k auf der rechten Seite einer Regel der Syntax eine neue (aktive/passive) Kante k'.

PROZEDUR $\text{REDUCE}_{\text{IS}}$

DATEN: Eine kontextfreie Syntax G = <V_N, V_T, S, R> und eine Chart C.
EINGABE: Eine passive Kante k = [i, j, A, e, α, e].

METHODE:
Für alle $B \rightarrow \tau \in R$, mit $\tau = \delta_1 \ldots \delta_n$ ($n \geq 1$):
Wenn <A = δ_i>
Dann <Trage [i, j, B, $\delta_1..\delta_{i-1}$, A, $\delta_{i+1} \ldots \delta_n$] in C ein>

Beispiel (8-4)
Wenn C eine Chart ist und G = <V_N, V_T, S, R> eine kontextfreie Syntax, mit
R = {S → B A B, S → A B A},
dann trägt $\text{REDUCE}_{\text{IS}}$ für die Kante [i, j, A, e, α, e] ∈ C folgende weitere Kanten in C ein:

[i, j, S, B, A, B],
[i, j, S, e, A, BA] und
[i, j, S, AB, A, e].

Mit diesen Operationen können wir jetzt einen Insel-Erkennungsalgorithmus formulieren.

ALGORITHMUS $\text{RECOGNIZE}_{\text{IS}}$

DATEN: Ein Lexikon L und eine kontextfreie Syntax G = <V_N, V_T, S, R> ohne Tilgungs- und Kettenregeln.
EINGABE: Ein Satz w = $w_1 \ldots w_n$ ($n \geq 1$).
AUSGABE: *True*/*False*.
ARBEITSSTRUKTUREN:

CHART	- Eine Chart. Anfangswert: leer.
INSELN	- Die Menge der Inseln von w. Anfangswert: leer.
INSEL-KATEGORIEN	- Die Menge von Kategorien, durch die die Inseln von w festgelegt werden.

METHODE:
INSEL-KATEGORIEN $\Leftarrow$ eine nicht-leere Teilmenge von V_T.
Initialisierung:
Für alle w_i $(1 \leq i \leq n)$:
Für alle lexikalischen Kategorien X, mit $w_i \in X$:
Wenn <X $\in$ INSEL-KATEGORIEN>
Dann <INSELN $\Leftarrow$ INSELN $\cup$ {[i-1, i, X, e, w_i, e]}>
Sonst <Chart $\Leftarrow$ Chart $\cup$ {[i-1, i, X, e, w_i, e]}>
Berechnung der übrigen Kanten:

Die Berechnung der Chart erfolgt, wie beim Earley-Algorithmus, in zwei Schritten:

- Initialisierung der Chart und
- Berechnung der übrigen Kanten.

Da nach der Initialisierung der Chart die Berechnung der anderen Kanten entweder durch Vorwärtsverkettung oder unter Zwischenspeicherung der neu generierten Kanten realisiert werden kann, spezifizieren wir beide Methoden separat.

Die Operationen $EXPAND_{IS}$, $COMPLETE_{IS}$ und $REDUCE_{IS}$ so zu modifizieren, daß sie verwendet werden können, um die Chart durch Vorwärtsverkettung bzw. Winograds Variante des Earley-Algorithmus zu berechnen, überlassen wir dem Leser und der Leserin als Übung. Zu beachten ist allerdings, daß $COMPLETE_{IS}$ nun in beiden Fällen so zu formulieren ist, daß eine passive oder aktive Kante als Argument akzeptiert wird, die dann mit allen geeigneten aktiven bzw. passiven Kanten der Chart kombiniert wird.

8.2.2 Berechnung der Chart durch Vorwärtsverkettung

Bei der Berechnung der Chart durch Vorwärtsverkettung wird jede neu gebildete Kante direkt zur Generierung weiterer Kanten verwendet. Den Kantenbildungsprozeß steuert die Prozedur **CLOSURE**$_{IS}$, die weitgehend der Prozedur CLOSURE aus Kapitel 7 entspricht. Den Algorithmus $RECOGNIZE_{IS}$ haben wir nur zu ergänzen durch:

Für jede Kante $k \in$ INSELN:
<$\underline{CLOSURE_{IS}(k)}$>
Wenn *<[0, n, S, e, α, e] $\in$ CHART>*
Dann *<RETURN(True)>*
Sonst *<RETURN(False)>.*

PROZEDUR CLOSURE$_{\text{IS}}$

DATEN: Eine kontextfreie Syntax G, ein Lexikon L und eine Chart C.
EINGABE: Eine Kante k = [i, j, A, α, β, γ].
SEITENEFFEKT: Berechnung der Chart.

METHODE:
Wenn <k $\notin$ C>
Dann <Trage k in C ein>
<$\underline{COMPLETE_{\text{IS}}}$(k)>
Wenn <k ist eine *aktive* Kante>
Dann <$\underline{EXPAND_{\text{IS}}}$(k)>
Sonst <$\underline{REDUCE_{\text{IS}}}$(k)>

8.2.3 Zwischenspeicherung der neu generierten Kanten

Wird zur Berechnung der Chart Winograds Variante des Earley-Algorithmus verwendet, werden neu gebildete Kanten zunächst zwischengespeichert. Wir verwenden zu diesem Zweck die Variable INSELN, die zunächst nur die Inseln des Satzes enthält.

Zur Berechnung der übrigen Kanten ist der Algorithmus wie folgt zu ergänzen:

Wenn $<S \rightarrow .\alpha. > \in$ *Chart(0,n)>*
Dann *<Return(True)>*
Wenn $<INSELN = \emptyset>$
Dann *<Return(False)>*
$K \Leftarrow$ *First(INSELN).*
$INSELN \Leftarrow$ *Rest(INSELN).*
$Chart \Leftarrow Chart \cup \{K\}$.
$INSELN \Leftarrow INSELN \cup COMPLETE_{\text{IS}}(K)$.
Wenn *<K ist eine aktive Kante>*
Dann $<INSELN \Leftarrow INSELN \cup EXPAND_{\text{IS}}(K)>$
Sonst $<INSELN \Leftarrow INSELN \cup REDUCE_{\text{IS}}(K)>$

Die Operationen EXPAND$_{\text{IS}}$, COMPLETE$_{\text{IS}}$ und REDUCE$_{\text{IS}}$ sind so zu formulieren, daß sichergestellt ist, daß keine Kanten generiert werden, die bereits in INSELN oder Chart enthalten sind.

Das abschließende Beispiel illustriert, in welchem Umfang die Wahl der Inseln den Analyseprozeß beeinflußt.

Beispiel (8-5)
Gegeben sei eine kontextfreie Syntax G = <V_N, V_T, S, R>, die die folgenden Regeln enthält:

S → NP VP	NP → det N1	NP → det n
NP → det adj n	N1 → adj n	VP → v adv.

Die Größe der Chart, die ein *depth-first* arbeitender Insel-Recognizer für den Satz „Der alte Mann starb heute." generiert, hängt stark von der Wahl der Insel-Kategorien ab:

(A) INSEL-KATEGORIEN[8]= {V}

1	*[0, 1, DET,* e, *DER,* e*]*		13	[2, 3, NP, DET, N, *e*]	C_{IS}
2	*[1, 2, A,* e, *ALTE,* e*]*		14	[2, 3, NP, DET A, N, *e*]	C_{IS}
3	*[2, 3, N,* e, *MANN,* e*]*		15	[2, 3, N1, A, N, *e*]	C_{IS}
4	*[4, 5, ADV,* e, *HEUTE,* e*]*		16	[1, 3, NP, DET, A N, *e*]	C_{IS}
5	**[3, 4, V,** *e*, **STARB,** *e*]		17	[1, 3, N1, *e*, A N, *e*]	C_{IS}
6	[3, 4, VP, *e*, V, ADV]	R_{IS}	18	[0, 3, NP, *e*, DET A N, *e*]	C_{IS}
7	[3, 5, VP, *e*, V ADV, *e*]	C_{IS}	19	[1, 3, NP, DET, N1, *e*]	C_{IS}
8	[3, 5, S, NP, VP, *e*]	R_{IS}	20	[0, 5, S, *e*, NP VP, *e*]	C_{IS}
9	[3, 3, NP, DET N1, *e*, *e*]	E_{IS}	21	[0, 3, S, *e*, NP, VP]	R_{IS}
10	[3, 3, NP, DET N, *e*, *e*]	E_{IS}	22	[0, 3, NP, *e*, DET N1, *e*]	C_{IS}
11	[3, 3, NP, DET A N, *e*, *e*]	E_{IS}	23	[3, 3, VP, *e*, *e*, V ADV]	E_{IS}
12	[3, 3, N1, A N, *e*, *e*]	E_{IS}			

(B) INSEL-KATEGORIEN = {det}

1	*[1, 2, A,* e, *ALTE,* e*]*		16	[0, 3, NP, *e*, DET N1, *e*]	C_{IS}
2	*[2, 3, N,* e, *MANN,* e*]*		17	[1, 3, NP, DET, N1, *e*]	C_{IS}
3	*[3, 4, V,* e, *STARB,* e*]*		18	[3, 4, VP, *e*, V, ADV]	C_{IS}
4	*[4, 5, ADV,* e, *HEUTE,* e*]*		19	[3, 5, VP, *e*, V ADV, *e*]	C_{IS}
5	**[0, 1, DET,** *e*, **DER,** *e*]		20	[0, 5, S, *e*, VP NP, *e*]	C_{IS}
6	[0, 1, NP, *e*, DET, A N]	R_{IS}	21	[3, 5, S, NP, VP, *e*]	C_{IS}
7	[0, 1, NP, *e*, DET, N]	R_{IS}	22	[3, 3, NP, DET N1, *e*, *e*]	E_{IS}
8	[0, 1, NP, *e*, DET, N1]	R_{IS}	23	[3, 3, NP, DET N, *e*, *e*]	E_{IS}
9	[0, 2, NP, *e*, DET A, N]	C_{IS}	24	[3, 3, NP, DET A N, *e*, *e*]	E_{IS}
10	[1, 1, N1, *e*, *e*, A N]	E_{IS}	25	[3, 3, N1, A N, *e*, *e*]	E_{IS}
11	[0, 3, NP, *e*, DET A N, *e*]	C_{IS}	26	[2, 3, NP, DET, N, *e*]	C_{IS}
12	[1, 2, N1, *e*, A, N]	C_{IS}	27	[2, 3, NP, DET A, N, *e*]	C_{IS}
13	[0, 3, S, *e*, NP, VP]	R_{IS}	28	[2, 3, N1, A, N, *e*]	C_{IS}
14	[1, 3, N1, *e*, A N, *e*]	C_{IS}	29	[1, 3, NP, DET, A N, *e*]	C_{IS}
15	[3, 3, VP, *e*, *e*, V ADV]	E_{IS}			

Wenn wir andere Kategorien bzw. Kombinationen von Kategorien als Insel-Kategorien auszeichnen, erhalten wir folgende Ergebnisse:

{N} bzw. {N, V}	-	23 Kanten
{A}	-	25 Kanten
{ADV}	-	25 Kanten
{DET}	-	29 Kanten
{DET, ADV}	-	31 Kanten
{DET, A, N, V}	-	31 Kanten

Wie sich zeigt, führt die Wahl *plausibler* Insel-Kategorien wie N oder V zu erheblich besseren Resultaten, als die Wahl anderer Kategorien (z.B. DET, ADV). Allerdings ist die Chart in keinem Fall kleiner als die Chart, die man durch Anwendung des Earley-Algorithmus erhalten hätte (23 Kanten). Vorteile können sich aber bei einem Parser ergeben, der nur den zuerst gefundenen Parse ausgibt; d.h. die Berechnung der Chart abbricht, sobald eine passive S-Kante für den Satz gefunden wurde.

8.3 Implementierung

Wir beschränken uns darauf, eine Lisp-Implementierung für einen Insel-Recognizer anzugeben, der die Chart unter Zwischenspeicherung der neu gebildeten Kanten berechnet (vgl. Kapitel 7.2.2).

8.3.1 Kantenselektoren

```
(defun start (kante)
  (first kante))
(defun ende (kante)
  (second kante))
(defun kopf (kante)
  (third kante))
(defun geschlossen (kante)
  (fifth kante))
(defun links (kante)
  (fourth kante))
(defun rechts (kante)
  (sixth kante))
```

AKTIVE-KANTE-P, PASSIVE-KANTE-P, NEUE-KANTE-P und MATCHING-KANTEN-P können unverändert aus Kapitel 7 übernommen werden.

8.3.2 Identifikation der Inseln im Satz

```
;;; LEXIKALISCHE-KANTEN
;;; Die Funktion generiert für ein Wort alle möglichen lexikalischen Kanten; d.h. eine
;;; für jede zugeordnete lexikalische Kategorie.
```

[8] Insel-Kanten sind durch Fettschrift hervorgehoben; bei der Initialisierung der Chart gebildete Kanten durch Kursivschrift. 'C_{IS}', 'E_{IS}' und 'R_{IS}' geben an, ob die Kante durch $COMPLETE_{IS}$, $EXPAND_{IS}$ oder $REDUCE_{IS}$ in die Chart eingetragen wurde.

```
(defun lexikalische-kanten (start wort)
  (mapcar #'(lambda (x) (list start (+ start 1) x nil (list wort) nil))
          (eintrag-kategorien wort)))
;;; INSELN
;;; Die Funktion legt durch einen mit dem Benutzer geführten Dialog fest, welche
;;; lexikalischen Kategorien als Insel verwendet werden. Sie berechnet die Liste der
;;; lexikalischen Kanten, die Inseln des Satzes repräsentieren (INSEL-KANTEN) und
;;; die lexikalischen Kanten, mit denen die Chart initialisiert wird (KANTEN).
(defun inseln (satz)
  (let ((z 0) (kategorien nil) (insel-kanten nil) (kanten nil))
    (format t "~%Welche der folgenden lexikalischen Kategorien: {~S }~%"
            (lexikalische-kategorien))
    (format t "sollen als Insel verwendet werden? ~%>  ")
    (setf kategorien (read))
    (dolist (x satz (list insel-kanten kanten))
      (dolist (y (lexikalische-kanten z x))
        (if (member (kopf y) kategorien)
            (push y insel-kanten)
            (push y kanten)))
      (setf z (+ z 1)))))
```

8.3.3 Die Hauptfunktionen

```
;;; RECOGNIZE-IS
;;; Top-level-Funktion des Programms.
(defun recognize-is (satz &key (symbol (startsymbol)))
  (do* ((l-kanten (inseln satz))
        (kante (caar l-kanten) (first p-kanten))
        (p-kanten (rest (first l-kanten)) (rest p-kanten))
        (chart (second l-kanten)))
       ((endp kante) (pprint chart) (auswerten chart symbol satz))
       (setf chart (update-chart chart kante)
             p-kanten (neue-kanten kante p-kanten chart))))

;;; NEUE-KANTEN
;;; Diese Funktion kombiniert P-Kanten mit den durch EXPAND-IS und
;;; COMPLETE-IS generierten neuen Kanten. Erweiterungen dieser Funktion er-
;;; lauben die Realisierung verschiedener Suchstrategien.
(defun neue-kanten (kante p-kanten chart)
  (let ((alte-kanten (append p-kanten chart)))
    (append p-kanten
            (remove-if #'(lambda (x) (member x alte-kanten :test #'equal))
```

```
                    (or (complete-is chart kante)
                        (if (aktive-kante-p kante)
                            (expand-is kante)
                            (reduce-is kante)))))))
;;; EXPAND-IS
;;; EXPAND-IS nimmt eine aktive Kante als Argument. Ist der rechte Abschnitt
;;; der Kante nicht leer, wird für jede Regel der Syntax, mit der das nächste Symbol
;;; dieses Abschnitts expandiert werden kann, eine neue Kante generiert; sonst für
;;; das letzte Symbol des rechten Abschnitts.
(defun expand-is (kante)
  (if (rechts kante)
      (mapcar #'(lambda (x) (list (ende kante) (ende kante) (first (rechts kante))
                                  nil nil x))
              (expansionen (first (rechts kante))))
      (mapcar #'(lambda (x) (list (start kante) (start kante)
                                  (first (last (links kante))) x nil nil))
              (expansionen (first (last (links kante)))))))

;;; REDUCE-IS
;;; Die Funktion nimmt als Argument eine passive Kante und generiert für jedes
;;; Vorkommen des Kopfs dieser Kante auf der rechten Seite einer der Regeln der
;;; Syntax eine aktive bzw. bei unären Regeln eine passive Kante.
(defun reduce-is (kante)
  (let ((resultat nil) (kat (kopf kante)))
    (dolist (x (regeln) resultat)
      (dotimes (y (length (rest x)))
        (when (eq (nth y (rest x)) kat)
          (push (list (start kante) (ende kante) (first x)
                      (butlast (rest x) (- (length (rest x)) y))
                      (list kat) (nthcdr (+ y 2) x)) resultat))))))

;;; COMPLETE-IS
;;; Die Funktion nimmt als Argument eine (aktive/passive) Kante und trägt in die
;;; Chart diejenigen Kanten ein, die sich durch Verbindung dieser Kante mit einer
;;; (passiven/aktiven) Kante der Chart bilden lassen.
(defun complete-is (chart kante)
  (if (passive-kante-p kante)
      (mapcar #'(lambda (x) (bilde-kante x kante))
              (remove-if #'(lambda (x)
                             (or (passive-kante-p x) (not (matching-kanten-p x kante))))
                         chart))
      (mapcar #'(lambda (x) (bilde-kante kante x))
```

```
                (remove-if #'(lambda (x)
                               (or (aktive-kante-p x) (not (matching-kanten-p kante x))))
                               chart))))
;;;; Kantenkonstruktor

;;; BILDE-KANTE
;;; Diese Funktion bildet aus den beiden Kanten KANTE1 und KANTE2 eine neue
;;; Kante: KANTE1 wird durch KANTE2 (rechts/links) ergänzt.
(defun bilde-kante (kante1 kante2)
  (if (rechts kante1)
      (list (start kante1)
            (ende kante2)
            (kopf kante1)
            (links kante1)
            (append (geschlossen kante1) (list (kopf kante2)))
            (rest (rechts kante1)))
      (list (start kante2)
            (ende kante1)
            (kopf kante1)
            (butlast (links kante1))
            (append (list (kopf kante2)) (geschlossen kante1))
            (rechts kante1))))
```

Die Funktionen AUSWERTEN und UPDATE-CHART sind mit den gleichnamigen Funktionen in Kapitel 7 identisch.

Aufgaben

8.1 Warum liefern die in Beispiel (8-5) verwendeten Insel-Hypothesen so unterschiedliche Ergebnissen?

8.2 Modifizieren Sie die Operation $\text{COMPLETE}_{\text{IS}}$ so, daß beidseitig erweiterbare Kanten gleichzeitig in beide Richtungen erweitert werden (vgl. 8.2).

8.3 Definieren Sie eine Prozedur, die es erlaubt, beliebige Wörter des zu analysierenden Satzes als Inseln zu markieren.

8.4 Erweitern Sie den Insel-Recognizer zu einem Insel-Parser, der Strukturbeschreibungen implizit (durch Verweislisten) repräsentiert. Es ist erforderlich, die Funktion GENERIERE-STRUKTUR aus Kapitel 7 für den Insel-Parser anzupassen.

8.5 Entwickeln Sie einen Insel-Parser, der eine Folge von Zeichen verarbeiten kann, bei der Wortgrenzen nicht durch Leerzeichen oder auf andere Weise markiert sind.

Kapitel 9

ID/LP-Syntaxen

In den letzten 10 Jahren hat sich ein deutlicher Wandel der Vorstellungen von der Struktur linguistischer Theorien und Formalismen vollzogen. Es galt eine Lösung für die fundamentalen methodologischen Probleme zu finden, zu der die Annahme einer hierarchischen Organisation der linguistischen Beschreibungsebenen, wie sie der amerikanische Strukturalismus populär gemacht hatte, geführt hatte. Neuere (meist unifikationsbasierte) Formalismen sind *modular* organisiert; d.h. sie postulieren eine Anzahl 'unabhängiger' Komponenten (Module), die relativ frei miteinander interagieren. Diesem Trend folgend, werden in verschiedenen Formalismen kontextfreie Phrasenstrukturregeln durch sogenannte *Immediate-dominance*- und *Linear-precedence*-Regeln (ID/LP-Regeln) ersetzt. Wir werden in diesem Kapitel zunächst das Konzept der ID/LP-Syntaxen erläutern[1] und anschließend zwei Varianten des Earley-Algorithmus vorstellen, die eine direkte Verarbeitung von ID/LP-Syntaxen ermöglichen.

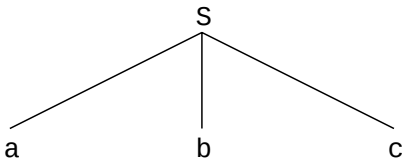
[1] Eine ausführliche Diskussion der linguistischen Adäquatheit von ID/LP-Syntaxen findet sich in [?].

9.1 Motivation für den ID/LP-Formalismus

Der ID/LP-Formalismus wurde ursprünglich im Rahmen der 'Generalisierten Phrasenstrukturgrammatik' (GPSG) entwickelt, und er wird heute in z.T. modifizierter Form auch in einer Reihe von anderen Formalismen (wie z.B. der *Head-Driven Phrase Structure Grammar*)[2] verwendet. Der wichtigste Grund für die Verwendung von ID- und LP-Regeln bildet die Tatsache, daß kontextfreie Regeln zwei unterschiedliche Typen von syntaktischen Relationen in einer Weise miteinander verknüpfen, die es unmöglich macht, wichtige Generalisierungen explizit auszudrücken. Da jede kontextfreie Regel gleichzeitig eine Dominanzbeziehung zwischen dem Symbol auf der linken und den Symbolen auf der rechten Regelseite, wie eine Abfolgebeziehung zwischen den Symbolen der rechten Regelseite determiniert, ist einer Grammatik, die Regeln dieses Typs verwendet, z.B. nicht direkt zu entnehmen, welche Abfolgerestriktionen *global*, d.h. für die gesamte Grammatik, gelten.

[2] Vgl. [?], [?] und [?].

Beispiel (9-1)
Die Regel S → a b c legt fest, daß S die Symbole a, b und c dominiert und daß a vor b und b vor c steht. Als Wohlgeformtheitsbedingung für lokale Bäume interpretiert, läßt sie genau einen Baum zu:



Diese Verknüpfung beider Relationen in einem Regeltyp ist aus linguistischer Sicht nur dann tolerierbar, wenn in der zu beschreibenden Sprache die Wortstellung eine bedeutende Aufgabe bei der Kodierung syntaktischer Informationen übernimmt (z.B. bei Sprachen mit relativ fester Wortstellung und wenig ausgeprägtem Flexionssystem wie dem Englischen). Bei Sprachen mit freier bzw. partiell freier Wortstellung dagegen verursacht sie eine erhebliche Redundanz der syntaktischen Beschreibung, die außerdem eine transparente Darstellung wichtiger Regularitäten verhindert.

Beispiel (9-2)
Nehmen wir einmal an, daß die folgende kontextfreie Syntax ein Fragment irgendeiner natürlichen Sprache vollständig und beobachtungsadäquat beschreibt:

$$S \rightarrow a\ b\ c \qquad S \rightarrow b\ c\ a \qquad S \rightarrow b\ a\ c$$

Das so definierte Sprachfragment läßt sich durch zwei Generalisierungen beschreiben, die als unabhängige *constraints* für die syntaktische Hierarchie bzw. lineare Abfolge aufgefaßt werden können:

1. Alle Konstituenten vom Typ S bestehen aus jeweils einem Vorkommen der Symbole a, b und c.

2. Das Symbol c darf nicht dem Symbol b vorangehen.

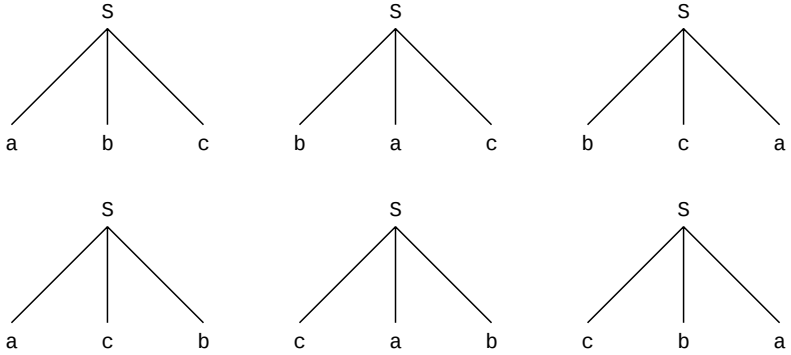
Diese Generalisierungen können in der Beschreibung des Fragments durch die kontextfreien Regeln nicht explizit ausgedrückt werden, da beide Relationen durch einen Regeltyp beschrieben werden.

In einer ID/LP-Syntax G = $<V_N, V_T, S, R_{ID}, R_{LP}>$ werden die Dominanz- und die Abfolgerelation jeweils durch einen eigenen Regeltyp spezifiziert:

Eine ID-Regel $X \rightarrow Y_1, \ldots, Y_n$ $(n \geq 1)$ legt fest, daß X die Symbole $Y_1, \ldots, Y_n$ dominiert, ohne gleichzeitig zu bestimmen, in welcher Reihenfolge die dominierten Symbole anzuordnen sind.

Diese Aufgabe übernehmen die LP-Regeln: Eine LP-Regel $Y_i < Y_j$ läßt nur solche Abfolgen zu, in denen Y_i vor Y_j steht[3]. Der Geltungsbereich einer LP-Regel ist die Menge der rechten Regelseiten aller ID-Regeln von G. So lassen sich durch sie für die ganze Syntax und die durch sie festgelegte Sprache geltende Ordnungsvorschriften ausdrücken[4].

Beispiel (9-3)
Die ID-Regel S → a, b, c läßt die folgenden sechs lokalen Bäume zu:



Zusammen mit der LP-Regel b < c, die nur die ersten drei der sechs lokalen Bäume zuläßt, erhält man eine zu der kontextfreien Syntax aus Beispiel (9-2) stark äquivalente ID/LP-Syntax, die außerdem beide Generalisierungen bzgl. der Dominanz- und Abfolgebeziehungen explizit ausdrückt.

Vergleicht man die generative Kapazität von kontextfreien Syntaxen mit der von ID/LP-Syntaxen, so läßt sich leicht zeigen, daß zwischen beiden Syntaxtypen folgende Beziehung besteht:

Zu jeder ID/LP-Syntax gibt es eine stark äquivalente kontextfreie Syntax, und umgekehrt kann für jede kontextfreie Syntax eine schwach äquivalente ID/LP-Syntax angegeben werden. Der ID/LP-Formalismus läßt sich also als eine Metasprache zur Beschreibung einer Teilklasse der kontextfreien Syntaxen betrachten. Daß nicht für jede, sondern nur für die kontextfreien Syntaxen, die die sogenannte ECPO-Eigenschaft („exhaustive constant partial ordering“[5]) aufweisen, eine stark äquivalente ID/LP-Syntax angegeben werden kann, läßt sich mit einem einfachen Beispiel gut illustrieren:

---

[3] '<' bezeichnet hier die lineare Präzedenzrelation, die als eine antisymmetrische und transitive Relation definiert ist.

[4] Formal kann die rechte Seite einer ID-Regel als eine Multimenge von Symbolen aufgefaßt werden; d.h. als eine Menge, in der ein Symbol mehrfach vorkommen kann.

[5] Vgl. [?, S.49f.]. Kurz gesagt besitzt eine kontextfreie Syntax die ECPO-Eigenschaft, wenn in ihr alle Abfolgerestriktionen *globale* Geltung besitzen; d.h. ihre Geltung nicht auf eine/einige Regeln beschränkt ist.

Beispiel (9-4)

Die durch LP-Regeln formulierten Abfolgerestriktionen gelten für die rechten Regelseiten aller ID-Regeln: Es ist nicht möglich, z.B. Restriktionen zu formulieren, die in einigen Regeln nur die Abfolge Y_1 Y_2, in anderen aber nur die Abfolge Y_2 Y_1 zulassen.

So gibt es z.B. für die Syntax, die nur die Regeln $X_1 \rightarrow Y_1\ Y_2$ und $X_2 \rightarrow Y_2\ Y_1$ enthält, keine stark äquivalente ID/LP-Syntax, da die Abfolge der dominierten Symbole von dem sie dominierenden Symbol abhängt. Eine Umformung in eine schwach äquivalente ID/LP-Syntax ist natürlich möglich, wenn auch relativ aufwendig:

$X_1 \rightarrow Z_1, Y_2$	$Z_1 \rightarrow Y_1$
$X_2 \rightarrow Z_2, Y_2$	$Z_2 \rightarrow Y_1$
$Z_1 < Y_2$	$Y_2 < Z_2$

Bevor wir auf die verschiedenen Möglichkeiten eingehen können, mit ID/LP-Syntaxen zu parsen, ist es erforderlich, den für ID/LP-Syntaxen geltenden Ableitungsbegriff zu entwickeln. Wir beginnen mit einer Reihe einfacher Definitionen:

Definition (9-1) *LP-zulässig*

Wenn $G = <V_N, V_T, S, R_{ID}, R_{LP}>$ eine beliebige ID/LP-Syntax ist und $w = w_1 \ldots w_n$ $(n \geq 1)$, mit $w_i \in V$, dann ist w LP-zulässig gdw. es für alle w_i $(1 \leq i < n)$ kein w_j $(i < j)$ gibt, so daß $w_j < w_i \in R_{LP}$.

Definition (9-2) *LP-Expansion*

Wenn $G = <V_N, V_T, S, R_{ID}, R_{LP}>$ eine beliebige ID/LP-Syntax ist und $w = w_1 \ldots w_n$ $(n \geq 1)$, mit $w_i \in V$, dann gilt: LP-Expansion(w) =

{w' | w' ist eine Permutation von w, und w' ist LP-zulässig}.

Beispiel (9-5)

Betrachten wir die ID/LP-Syntax aus Beispiel (9-3), dann gilt für w = abc:

LP-Expansion(abc) = {abc, bac, bca}.

Definition (9-3) *(direkte) Ableitbarkeit*

Wenn $G = <V_N, V_T, S, R_{ID}, R_{LP}>$ eine beliebige ID/LP-Syntax ist, dann ist $w = w_1 \ldots w_n$ $(n \geq 1)$, mit $w_i \in V$, aus $A \in V_N$ *direkt ableitbar* $(A \Rightarrow_{ID} w)$ gdw. $A \rightarrow w' \in R_{ID}$ und $w \in$ LP-Expansion(w').

Der reflexive, transitive Abschluß dieser Relation wird mit '$^* \Rightarrow_{ID}$' bezeichnet.

Wie bei kontextfreien Syntaxen besteht die durch eine ID/LP-Syntax G festgelegte Sprache aus der Menge aller Ketten über V_T, die aus dem Startsymbol von G ableitbar sind.

9.2 Parsen mit ID/LP-Syntaxen

Es gibt verschiedene Strategien, die beim Parsen mit ID/LP-Syntaxen verwendet werden können. Die beiden Eckpunkte der Skala möglicher Verarbeitungsstrategien bilden die beiden Strategien, die als *direktes Parsen* bzw. *indirektes Parsen* von ID/LP-Syntaxen bezeichnet werden.

9.2.1 Indirektes Parsen

Da es zu jeder ID/LP-Syntax G = <V_N, V_T, S, R_{ID}, R_{LP}> eine stark äquivalente kontextfreie Syntax gibt, ist es möglich, für das Parsen nicht die ID/LP-Syntax zu verwenden, sondern die zu G stark äquivalente Syntax G‘, die sich durch Berechnung der LP-zulässigen Permutationen der rechten Seiten der ID-Regeln von G ergibt. Der Vorteil dieses zweistufigen Verfahrens liegt darin, daß zum Parsen die für kontextfreie Syntaxen verfügbaren Algorithmen ohne Modifikation übernommen werden können. Diesem Vorteil steht allerdings ein gravierender Nachteil gegenüber: Die Expansion einer ID/LP-Syntax zu einer kontextfreien Syntax kann die Zahl der Regeln (kontextfreie versus ID-Regeln) äußerst stark anwachsen lassen. Im günstigsten Fall, wenn für jede ID-Regel jeweils nur genau eine Permutation der Symbole der rechten Regelseite zulässig ist, bleibt die Zahl der kontextfreien Regeln gleich der Zahl der ID-Regeln. Andererseits ist es aber auch möglich, daß die Menge der LP-Regeln leer ist; d.h. daß jede Permutation zulässig ist. In diesem Fall ist die Zahl der kontextfreien Regeln, die für eine ID-Regel $A \rightarrow \alpha$ erzeugt werden, gleich der Fakultät von $|\alpha|$.

Beispiel (9-6)
Aus der ID-Regel S → a, b, c werden sechs kontextfreie Regeln erzeugt:

S → a b c	S → b a c	S → b c a
S → a c b	S → c a b	S → c b a.

Berücksichtigt man darüber hinaus, daß heute Grammatikformalismen häufig als komplexe Systeme von interagierenden Modulen konzipiert sind, die eine Expansion in eine kontextfreie Syntax (wenn sie überhaupt möglich ist) erheblich erschweren, scheint die Trennung von einem (linguistischen Einsichten folgenden) 'Beschreibungsformalismus' und einem 'Verarbeitungsformalismus' nicht immer angebracht[6].

[6] Außerdem hat die Zahl der Regeln einer Grammatik entscheidenen Einfluß auf die Verarbeitungszeit; vgl. [?].

9.2.2 Direktes Parsen

Für eine direkte Verarbeitung von ID/LP-Syntaxen, die eine Expansion in eine stark äquivalente kontextfreie Syntax überflüssig macht, ist es erforderlich, die gängigen Parsingalgorithmen zu modifizieren. Wir werden uns auf S.Shiebers und G.E.Bartons Modifikation des Earley-Algorithmus beschränken.

Um mit diesem Algorithmus ID/LP-Syntaxen direkt verarbeiten zu können, sind zwei Typen von Änderungen notwendig:

1. eine Änderung der Objekte, die in der Chart gespeichert werden, und
2. eine Änderung der Operationen, durch die diese Objekte berechnet werden.

Die beiden Modifikationen des Earley-Algorithmus, die wir betrachten werden, unterscheiden sich nur hinsichtlich des ersten Punktes.

Shiebers Algorithmus

Alle Kanten, die in einer durch den Earley-Algorithmus berechneten Chart enthalten sind, korrespondieren direkt mit den Regeln der Syntax (bzw. den Einträgen des Lexikons):
Für jede Kante [i, j, A, α, β] aus der Chart C gibt es in der Syntax eine Regel (bzw. im Lexikon einen Eintrag) der Form $A \rightarrow \alpha\beta$. Formal besteht der Unterschied zwischen ID-Regeln und kontextfreien Regeln darin, daß die rechte Regelseite nicht eine Kette, sondern eine Multimenge von Symbolen ist.

Dieser Unterschied muß sich auch in den von einem earley-basierten Parser für ID/LP-Syntaxen generierten Kanten widerspiegeln: Zumindest der offene Abschnitt der Kanten muß ebenfalls als Multimenge interpretiert werden, da jedes Symbol des offenen Abschnitts einen möglichen Kandidaten für die Fortsetzung der Analyse repräsentiert. Der geschlossene Abschnitt dagegen kann weiterhin als Kette aufgefaßt werden, da er eine der zulässigen Linearisierungen des erkannten Satzabschnitts repräsentiert. Diese Lösung wählt Shieber: Eine Kante hat in seiner Variante des Earley-Algorithmus die Form [i, j, A, α, $\{B_1, \ldots, B_n\}_M$], mit $n \geq 0$.

Die Modifikation der Operationen muß sicherstellen, daß die Informationen, die ID- und LP-Regeln separat bereitstellen, bei der Berechnung der Chart C in adäquater Form kombiniert werden[7].

[7] Wir orientieren uns bei der Darstellung der folgenden beiden Operationen an der ursprünglichen Fassung der Operationen EXPAND und COMPLETE aus Kapitel 7. Wir setzen allerding voraus, daß $\text{EXPAND}_{\text{ID/LP}}$ infolge der Trennung von Syntax und Lexikon nur auf syntaktischen Kategorien operiert, da so eine Änderung der Prozedur SHIFT überflüssig wird.

(1) $\text{EXPAND}_{\text{ID/LP}}$

Wenn [i, j, A, α, $\{B_1, \ldots, B_n\}_M$] $\in$ C, dann muß für alle B_i, die als erstes Symbol einer LP-Expansion von $\{B_1, \ldots, B_n\}_M$ vorkommen, für jede Regel $B_i \rightarrow \tau$ eine Kante [j, j, B_i, e, τ] in C eingetragen werden:

PROZEDUR $\text{EXPAND}_{\text{ID/LP}}$

DATEN: Eine ID/LP-Syntax G = <V_N, V_T, S, R_{ID}, R_{LP}> und eine Chart C.

METHODE:

Wenn <[i, j, A, α, $\{B\}_M \cup_M \{\beta\}_M$] $\in$ C, mit $(0 \leq i \leq j \leq n)$ und
B ist das erste Symbol einer LP-Expansion von $\{B\}_M \cup_M \{\beta\}_M$ >

Dann <Für alle $B \rightarrow \tau \in R_{ID}$:
<Trage [j, j, B, e, τ] in C ein>>

(2) $\text{COMPLETE}_{\text{ID/LP}}$

Wenn [i, j, A, α, $\{B_1, \ldots, B_n\}_M$] $\in$ C, dann muß für jede Kante [j, l, B_i, τ, e] $\in$ C, mit B_i als erstem Symbol einer LP-Expansion von $\{B_1, \ldots, B_n\}_M$, eine Kante [i, l, A, αB_i, $\{B_1, \ldots, B_{i1}, B_{i+1}, \ldots, B_n\}_M$] in die Chart C eingetragen werden:

PROZEDUR $\text{COMPLETE}_{\text{ID/LP}}$

DATEN: Eine ID/LP-Syntax G = <V_N, V_T, S, R_{ID}, R_{LP}> und eine Chart C.

METHODE:

Wenn <[i, j, A, α, $\{B\}_M \cup_M \{\beta\}_M$] $\in$ C und [j, l, B, τ, e] $\in$ C, mit B als erstem Symbol einer LP-Expansion von $\{B\}_M \cup \{\beta\}$ und $0 \leq i \leq j \leq l \leq n$>

Dann <Trage [i, l, A, αB, $\{\beta\}_M$] in C ein>

Beispiel (9-7)

C sei eine Chart, und G sei eine ID/LP-Syntax mit den folgenden ID- und LP-Regeln:

$S \rightarrow A, S, B$
$S \rightarrow A, B$ $\qquad\qquad$ $A < B$
$A \rightarrow a, a$
$B \rightarrow b, b$

Wenn [i, i, S, e, $\{A, S, B\}_M$] $\in$ C, dann trägt $\text{EXPAND}_{\text{ID/LP}}$ auch die Kanten [i, i, A, e, $\{a, a\}_M$] und [i, i, S, e, $\{A, B\}_M$], nicht aber die Kante [i, i, B, e, $\{b, b\}_M$] in C ein.

Wenn [i, i, S, e, $\{A, S, B\}_M$], [i, j, S, A B, e] und [i, j, A, a a, e] $\in$ C $(j \geq i)$, dann generiert $\text{COMPLETE}_{\text{ID/LP}}$ die Kanten: [i, j, S, S, $\{A, B\}_M$] und [i, j, S, A, $\{S, B\}_M$].

Bartons Repräsentation

Bartons Repräsentation der Kanten unterscheidet sich von der Shiebers nur darin, daß er auch den geschlossenen Abschnitt einer Kante als Multimenge repräsentiert[8]. Diese Repräsentation erweist sich als vorteilhaft, wenn hochgradig ambige Syntaxen zu verarbeiten sind, bzw. bei starker lexikalischer Ambiguität, wie das folgende Beispiel aus [**?**, **?**] deutlich zeigt:

Beispiel (9-8)
Gegeben sei eine ID/LP-Syntax G mit den folgenden ID-Regeln ($R_{LP} = \emptyset$):

S → A, B, C, D, E
A → a | x
B → b | x
C → c | x
D → d | x
E → e | x

Daß die Zahl der bei der Analyse eines Satzes generierten Kanten abhängig von der für die Kanten verwendeten Repräsentation stark differiert, ist offensichtlich:
Für den Satz xxxxa werden 251 (Shieber) bzw. 53 (Barton) Kanten generiert.

Anzahl der Kanten

	Shieber	Barton	indirektes Parsen
Chart(0,0)	1	1	120
Chart(0,1)	5 (5)[9]	5 (5)	120 (5)
Chart(0,2)	20	10	120
Chart(1,2)	(5)	(5)	(5)
Chart(0,3)	60	10	120
Chart(2,3)	(5)	(5)	(5)
Chart(0,4)	120	5	120
Chart(3,4)	(5)	(5)	(5)
Chart(0,5)	24	1	24
Chart(4,5)	(1)	(1)	(1)
Summe	251	53	645

Die Repräsentation Bartons ist 'kompakter', da Informationen, die bei Verwendung von Shiebers Repräsentation durch mehrere Kanten dargestellt werden, nun durch eine einzige Kante ausgedrückt werden können.

[8] Barton führt seine Repräsentation in zwei Aufsätzen ein, in denen er zeigt, daß Shiebers Einschätzung der Komplexität seines Algorithmus bei weitem zu optimistisch war: ID/LP-Parsing ist entgegen Shiebers Annahme NP-vollständig.

[9] Die in Klammern gesetzten Zahlen geben die Zahl der *passiven* Kanten an.

Beispiel (9-9)
Legt man Syntax und Satz des letzten Beispiels zugrunde, enthält Chart C nach der Verarbeitung der ersten drei Worte bei Verwendung von Shiebers Kantenrepräsentation u.a. die folgenden Kanten:

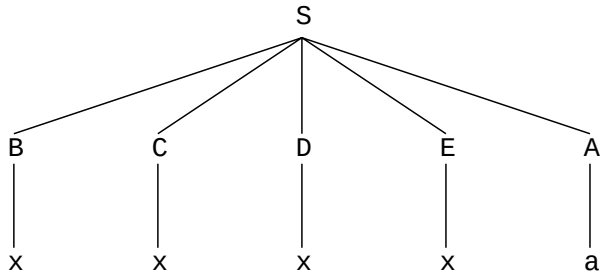
[0, 3, S, ABC, {D,E}]	[0, 3, S, ACB, {D,E}]	[0, 3, S, CAB, {D,E}]
[0, 3, S, CBA, {D,E}]	[0, 3, S, BAC, {D,E}]	[0, 3, S, BCA, {D,E}]

Verwendet man Bartons Repräsentation, enthält die Chart statt dieser 6 Kanten nur die Kante [0, 3, S, {A,B,C}, {D,E}].

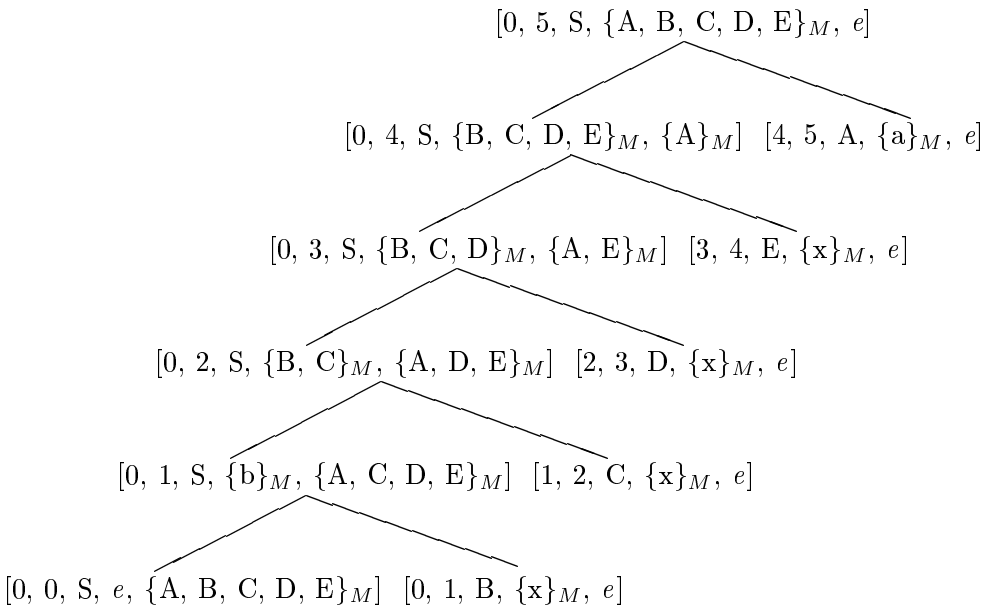
Allerdings kann Bartons Repräsentation insofern zu Übergeneralisierungen führen, als dem geschlossenen Abschnitt einer Kante nun nicht mehr zu entnehmen ist, welche Linearisierungen der in ihm enthaltenen Symbole zulässig ist: Wenn wir die Syntax aus Beispiel (9-8) durch die LP-Regel A < B ergänzen, enthält die Chart bei Shiebers Repräsentation nur noch die ersten drei Kanten, bei Bartons Repräsentation natürlich weiterhin die Kante [0, 3, S, {A,B,C}, {D,E}].

Bleibt noch die Frage, wie beim direkten Parsen von ID/LP-Syntaxen die Generierung der Strukturbeschreibungen gestaltet werden sollte. Eine effiziente Kantenrepräsentation sollte mit einer möglichst effizienten Repräsentation der Strukturbeschreibungen kombiniert werden. Damit scheidet eine explizite Speicherung der Strukturbeschreibungen innerhalb der Kanten aus. Ein geeignetes Verfahren ist die implizite Repräsentation der Strukturbeschreibungen durch Verweispaare nach [**?**].

Beispiel (9-10)
Zu den Strukturbeschreibungen, die für den Satz xxxxa relativ zur Syntax aus Beispiel (9-8) gebildet werden können, gehört auch die Struktur



Mit ihr korrespondiert bei einer impliziten Repräsentation der Strukturbeschreibung durch Verweispaare die Folge von Verweispaaren, die die folgende 'Bildungsgeschichte' für die Kante [0, 5, S, {A,B,C,D,E}, e] abbildet:



9.3 Implementierung

9.3.1 Lisp

Da nur geringe Änderungen erforderlich sind, um die Implementierung des Earley-Algorithmus aus Kapitel 7.2.1 (Version: Berechnung der Chart durch Vorwärtsverkettung) so zu modifizieren, daß ID/LP-Syntaxen korrekt verarbeitet werden, geben wir nur neu hinzugekommene bzw. geänderte Funktionen an.

Spezielle Funktionen zum Verarbeiten von ID/LP-Syntaxen

```
;;; ZULAESSIGE-SYMBOLE
;;; Diese Funktion stellt fest, welche Symbole aus LISTE den anderen Symbolen aus
;;; LISTE vorangehen können, ohne daß eine der LP-Regeln der Syntax verletzt wird.
(defun zulaessige-symbole (liste)
  (let ((resultat nil))
    (dolist (x (remove-duplicates liste) resultat)
      (when (kann-vorangehen-p x (waehle x liste)) (push x resultat)))))

;;; KANN-VORANGEHEN
;;; Die Funktion prüft, ob SYMBOL den Symbolen aus LISTE vorangehen kann,
;;; ohne daß eine der LP-Regeln verletzt wird.
(defun kann-vorangehen-p (symbol liste)
  (dolist (x liste t)
```

```
    (when (member (list x symbol) (lp-regeln) :test #'equal) (return nil))))
;;; WAEHLE
;;; Das erste Vorkommen von OBJEKT aus LISTE wird entfernt.
(defun waehle (object liste)
  (remove object liste :count 1))
```

Geänderte Funktionen

```
;;;; Neue Selektoren für ID/LP-Syntaxen:
(defun id-regeln ()              (defun lp-regeln ()
  (fourth *syntax*))               (fifth *syntax*))
```

Die wichtigsten Änderungen[10] betreffen die Funktionen, die neue Kanten generieren:

- Bei der Funktion EXPAND, die zyklische Kanten generiert, muß berücksichtigt werden, daß jedes Symbol des offenen Abschnitts einer Kante, das den anderen vorangehen kann, zur Bildung neuer zyklischer Kanten führt.
- Bei der Bildung neuer Kanten durch die Verbindung einer aktiven mit einer passiven Kante durch COMPLETE gilt es zu beachten, daß jede passive Kante, deren Kopf im offenen Abschnitt der aktiven Kante enthalten ist, hierfür ein möglicher Kandidat ist.

```
;;; EXPAND
;;; Die Funktion nimmt eine aktive Kante als Argument und generiert für jede Regel
;;; der Syntax, mit der eines der legalen Symbole (s. ZULAESSIGE-SYMBOLE) des
;;; offenen Abschnitts der Kante expandiert werden kann, eine neue Kante.
(defun expand (kante)
  (mapcar #'(lambda (x) (list (ende kante) (ende kante) (first x) nil (rest x)))
            (zyklische-kanten (offen kante))))

;;; ZYKLISCHE-KANTEN
;;; Die Funktion berechnet auf der Grundlage der zulässigen Symbole des offenen Ab-
;;; schnitts einer aktiven Kante und der Regeln der Syntax die Menge aller möglichen
;;; zyklischer Kanten.
(defun zyklische-kanten (open)
  (let ((symbole (zulaessige-symbole open)) (resultat nil))
    (dolist (x symbole resultat)
      (setf resultat (append resultat (matching-regeln x))))))

;;; MATCHING-REGELN
;;; Sie generiert alle ID-Regeln, deren linke Seite mit SYMBOL identisch ist.
```

[10] Änderungen werden durch **Fettschrift** hervorgehoben.

```
(defun matching-regeln (symbol)
  (remove-if-not #'(lambda (x) (eq symbol (first x))) (id-regeln)))
;;; COMPLETE
;;; Die Funktion nimmt als Argument eine (aktive/passive) Kante und trägt in die
;;; Chart diejenigen Kanten ein, die sich durch Verbindung dieser Kante mit einer
;;; (passiven/aktiven) Kante der Chart bilden lassen.
(defun complete (chart kante)
  (if (passive-kante-p kante)
      (mapcar #'(lambda (x) (bilde-kante x kante))
          (remove-if #'(lambda (x)
                       (or (passive-kante-p x) (not (matching-kanten-p x kante))))
                       chart))
      (mapcar #'(lambda (x) (bilde-kante kante x))
          (remove-if #'(lambda (x)
                       (or (aktive-kante-p x) (not (matching-kanten-p kante x))))
                       chart))))

;;; MATCHING-KANTEN-P
;;; Das Prädikat testet, ob KANTE1 mit KANTE2 verbunden werden kann.
(defun matching-kanten-p (kante1 kante2)
  (and (eql (ende kante1) (start kante2))
       (member (kopf kante2) (zulaessige-symbole (offen kante1)))))

;;; BILDE-KANTE
;;; Aus KANTE1 und KANTE2 wird eine neue Kante gebildet.
(defun bilde-kante (kante1 kante2)
  (list (start kante1) (ende kante2) (kopf kante1)
        (append (geschlossen kante1) (list (kopf kante2)))
        (waehle (kopf kante2) (offen kante1))))
```

Aufgaben

9.1 Schreiben Sie ein Programm, das eine ID/LP-Syntax in eine stark äquivalente kontextfreie Syntax konvertiert.

9.2 Schreiben Sie ein Programm, das prüft, ob eine kontextfreie Syntax die ECPO-Eigenschaft besitzt und sie, wenn diese Bedingung erfüllt ist, in eine stark äquivalente ID/LP-Syntax konvertiert.

9.3 Welche Sprache generiert die Syntax aus Beispiel (9-7)?

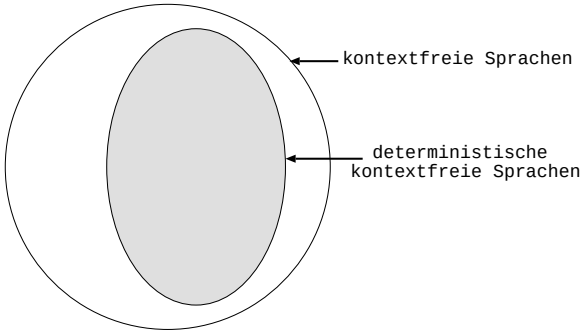
9.4 Schreiben Sie einen Parser für ID/LP-Syntaxen, der mit Bartons Repräsentation arbeitet.

Teil IV

Deterministisches Parsen

In den ersten beiden Teilen des Buchs haben wir Parsingalgorithmen beschrieben, die kontextfreie Syntaxen verarbeiten können, die die Generierung beliebiger kontextfreier Sprachen erlauben. Diese Algorithmen verwendeten bei der Suche nach einer Ableitung für einen gegebenen Satz verschiedene Strategien: Depth-first-Suche mit Backtracking, Breadth-first-Suche ohne Backtracking oder die Speicherung aller Teilergebnisse in einer Chart.

Betrachtet man die Menge der kontextfreien Sprachen $\mathcal{L}$, dann gibt es eine Menge von *deterministischen kontextfreien Sprachen* $\mathcal{L}_{\text{det}}$, mit $\mathcal{L}_{\text{det}} \subset \mathcal{L}$[11]. Die Menge $\mathcal{L}_{\text{det}}$ enthält alle kontextfreien Sprachen, für die ein deterministischer Kellerautomat konstruiert werden kann, der sie akzeptiert.



Für deterministische kontextfreie Sprachen lassen sich Syntaxen formulieren, die die Verwendung von Erkennungs- und Parsingalgorithmen erlauben, die sich durch eine erheblich günstigere Zeit- und Raumkomplexität gegenüber den vergleichbaren Algorithmen für generelle kontextfreie Sprachen auszeichnen. Diese *deterministischen Parsingalgorithmen* sind vor allem im Compilerbau von Bedeutung, da viele Programmiersprachen in diese Klasse fallen.

Für die Computerlinguistik sind diese deterministische Parsingalgorithmen von relativ untergeordneter Bedeutung; denn obwohl die Debatte über die Kontextfreiheit natürlicher Sprachen nach mehr als 30 Jahren immer noch kein Ende gefunden hat, galt und gilt es als unstrittig, daß natürliche Sprachen nicht als deterministische kontextfreie Sprachen betrachtet werden können. Da aber der Anfang der 80er Jahre entwickelte Algorithmus von M.Tomita, der in zahlreichen natürlichsprachlichen Systemen verwendet wird, auf einem Parsingalgorithmus für LR-Syntaxen, einem speziellen Typ deterministischer kontextfreier Syntaxen, basiert[12], erläutern wir in Kapitel 10 zunächst die Begriffe „LR-Syntax“ bzw. „LR -Sprache“ und formulieren einen einfachen LR-Parsingalgorithmus, bevor wir in Kapitel 11 den Tomita-Algorithmus beschreiben.

[11] Die Sprache $L = \{0^n 1^n \mid n \geq 1\} \cup \{0^n 1^{2n} \mid n \geq 1\}$ ist ein Beispiel für eine Sprache, die in $\mathcal{L}$ enthalten ist, nicht aber in $\mathcal{L}_{\text{det}}$.

[12] Der Tomita-Algorithmus kann als ein *generalisierter* LR-Parsingalgorithmus betrachtet werden, der die Verarbeitung beliebiger kontextfreier Syntaxen erlaubt.

Kapitel 10

LR-Parsing

LR-Parser gehören zu der Klasse der deterministischen Shift-reduce-Parser[1]. Wir charakterisieren zunächst die Klasse der Syntaxen, die sich mit LR-Parsern verarbeiten lassen. Dann beschreiben wir den Aufbau und die Arbeitsweise von LR-Parsern und geben schließlich einen Algorithmus zur Generierung von LR(1)-Parsingtabellen an.

10.1 LR-Syntaxen

Die Klasse der Syntaxen, die von LR-Parsern verarbeitet werden können, werden als „LR-Syntaxen“ bezeichnet. Wie alle Shift-reduce-Parser verwenden auch LR-Parser zwei Prozeduren, die sich als Stackoperationen auffassen lassen (vgl. Kapitel 4): Entweder wird der den aktuellen Stand der Analyse repräsentierende Stack durch Anwendung einer Regel reduziert (*reduce*) oder das nächste Wort des Satzes auf den Stack geschoben (*shift*).
Abhängig von der Länge des Look-ahead, d.h. von der Zahl der Symbole des noch nicht analysierten Satzabschnitts, die betrachtet werden müssen, um zu entscheiden, mit welcher Regel im nächsten Schritt der Stack zu reduzieren ist bzw. ob das nächste Wort des Satzes auf den Stack zu schieben ist, unterscheidet man verschiedene Typen von LR-Syntaxen:

LR(0)-Syntaxen, LR(1)-Syntaxen, ..., LR(k)-Syntaxen.

Aus Gründen der Effizienz beschränkt man sich normalerweise auf die Betrachtung von LR(0)- bzw. LR(1)-Syntaxen[2].

[1] Sie werden „LR-Parser“ genannt, weil sie Eingaben von **l**inks nach rechts verarbeiten und **R**echtsreduktionen generieren.

[2] Außerdem ist es möglich, für jede LR(k)-Syntax eine schwach äquivalente LR(0)-Syntax zu konstruieren, die dann allerdings eine erheblich größere Zahl von nicht-terminalen Symbolen enthält.

Die LR(k)-Syntaxen bilden die größte Klasse von kontextfreien Syntaxen, die sich mit deterministischen Bottom-up-Parsern verarbeiten lassen. Die Klasse der kontextfreien Sprachen, für die es eine LR(k)-Syntax gibt, ist extensional äquivalent mit der Klasse von kontextfreien Sprachen, die von deterministischen Kellerautomaten akzeptiert werden.

Ein Shift-reduce-Parser erzeugt bei der Analyse eines Satzes $w = w_1 \ldots w_n$ $(n \geq 1)$ eine Rechtsreduktion für w: Wenn $G = <V_N, V_T, S, R>$ eine kontextfreie Syntax ist und $<\alpha_0, \alpha_1, \ldots, \alpha_m>$ eine Rechtsableitung für w in G ist, dann ist $<\alpha_m, \alpha_{m-1}, \ldots, \alpha_1, \alpha_0>$ eine Rechtsreduktion von w. Jedes α_i wird als „Rechtssatzform" bezeichnet und hat die Form βx, mit $\beta = e$ oder β endet mit einem nicht-terminalen Symbol und $x \in V_T^*$. β wird als „offener Teil" von α_i und x als „geschlossener Teil" von α_i bezeichnet.

Wenn ein Shift-reduce-Parser sich in einem einer Rechtssatzform α_i korrespondierenden Zustand befindet, dann enthält der Stack den *offenen Teil* von α_i, und der *geschlossene Teil* entspricht der noch nicht verarbeiteten Eingabe; anders formuliert: Mit der Rechtsableitung korrespondiert die inverse Folge von Reduce-Operationen, die der Parser ausführt. Zwischen zwei Reduce-Operationen kann jeweils eine unbestimmte Zahl von Shift-Operationen (≥ 0) erforderlich sein.

Bei einer LR(0)-Syntax ist es allein aufgrund des Stackinhalts (*offener Teil* der korrespondierenden Rechtssatzform) möglich zu entscheiden, welche Regel für den nächsten Reduktionsschritt zu verwenden ist. Dieser Sachverhalt läßt sich mit Hilfe des Begriffs des *linken Kontextes* eines Symbols bzw. einer Regel präzise formulieren[3]. Wenn $G = <V_N, V_T, S, R>$ eine kontextfreie Syntax ist, dann gilt:

Definition (10-1) *Linker Kontext eines Symbols*

Für alle $X \in V$ ist der linke Kontext - notiert als „LK(X)" - definiert als:

$$LK(X) = \{w \mid S \underset{rm}{\overset{*}{\Longrightarrow}} wXz, \text{ mit } w, z \in V^*\}.$$

Der linke Kontext eines Symbols besteht also aus der Menge aller Ketten, die in einer Rechtssatzform von G links von X vorkommen können.

Definition (10-2) *Linker Kontext einer Regel*

Für alle $X \to w \in R$ ist der linke Kontext - notiert als „LK($X \to w$)" - definiert als:

$$LK(X \to w) = LK(X) * \{w\}.$$

Der linke Kontext einer Regel besteht aus dem Produkt des linken Kontextes der linken Regelseite und der rechten Regelseite.

Definition (10-3) *LR(0)-Syntax*

[3] Wir folgen in unserer Darstellung [?, S.134-39].

G ist eine LR(0)-Syntax, wenn für alle paarweise verschiedenen Regeln X→ w, X' → w' ∈ R gilt:

$$LK(X \to w) \cap INIT(LK(X' \to w')) = \emptyset$$[4].

G ist eine LR(0)-Syntax gdw. für beliebige Regeln r_1, r_2 gilt: Keine Kette des Linkskontextes von r_1 ist ein Anfangssegment einer Kette aus dem Linkskontext von r_2.

Beispiel (10-1)

Für die Syntax $G_1 = <\{S, C, D\}, \{a, b, c\}, S, \{S \to C, S \to D, C \to aC, C \to b, D \to aD, D \to c\}>$ gilt:

(a) $LK(C) = \{a^n | \ n \geq 0\}$;

(b) $LK(C \to aC) = a^n aC$ und

(c) G_1 ist eine LR(0)-Syntax; denn

$LK(S \to C) = \{C\}$ $\quad$ $LK(S \to D) = \{D\}$

$LK(C \to aC) = \{a^n aC | \ n \geq 0\}$ $\quad$ $LK(C \to b) = \{a^n b | \ n \geq 0\}$

$LK(D \to aD) = \{a^n aD | \ n \geq 0\}$ $\quad$ $LK(D \to c) = \{a^n c | \ n \geq 0\}$.

Bei einer LR(k)-Syntax muß allein anhand des *offenen Teils* einer Rechtssatzform und der ersten k Symbole des *geschlossenen Teils* entschieden werden können, wie diese Rechtssatzform zu reduzieren ist.

Definition (10-4) *LR(k)-Syntax*

Eine kontextfreie Syntax $G = <V_N, V_T, S, R>$ ist eine LR(k)-Syntax ($k \geq 0$) gdw. für die *erweiterte Syntax*[5] $G' = <V_N \cup \{S'\}, V_T, S', R \cup \{S' \to S\}>$ gilt, daß immer, wenn es zwei Rechtsableitungen der folgenden Form gibt

$$S' \underset{G'_{*rm}}{\overset{*}{\Longrightarrow}} \alpha A w_{\,rm} \Longrightarrow \alpha\beta w$$

$$S' \underset{G'\,rm}{\Longrightarrow} \gamma B x_{\,rm} \Longrightarrow \alpha\beta y$$

mit $FIRST_k(w) = FIRST_k(y)$,[6] daraus folgt, daß $\alpha Ay = \gamma Bx$;
d.h. $\alpha = \gamma$, $A = B$ und $x = y$.

Wenn also $\alpha\beta w$ und $\alpha\beta y$ zwei Rechtssatzformen von G' sind und $A \to \beta$ die Regel ist, die im letzten Schritt der Ableitung von $\alpha\beta w$ verwendet wurde, dann muß diese Regel auch in der Reduktion von $\alpha\beta y$ auf αAy verwendet werden.

Beispiel (10-2)

Die Syntax $G_2 = <\{S, A, B, C, D, E\}, \{a, b\}, S, R>$, mit

$$R = S \to AB \quad A \to a \quad B \to CD \quad B \to aE$$

[4] Mit „INIT(α)“ wird die Menge aller Anfangssegmente der Ketten aus α bezeichnet: INIT({a, abc}) = {a, ab, abc}.

[5] Die Einführung des neuen Startsymbols und der neuen Regel hat zur Folge, daß ein Satz akzeptiert wird gdw. der LR-Parser diese Regel für einen Reduktionsschritt verwendet (vgl. [?, S.372]).

[6] „$FIRST_K(w)$“ bezeichnet die Kette, die aus den ersten k Symbolen von w besteht.

$$C \to ab \qquad D \to bb \quad E \to bba$$

ist keine LR(0)-Syntax, denn es gilt:

$$LK(A \to a) \cap LK(B \to CD) = \{a\} \cap INIT\{ACD, aCD\} \neq \emptyset.$$

Damit ist gezeigt, daß G_2 eine LR(2)-Syntax ist.

Wenn „$\mathcal{L}_{LR(0)}$“/„$\mathcal{L}_{LR(k)}$“ die Menge aller Sprachen bezeichnet, die sich durch eine LR(0)-Syntax bzw. LR(k)-Syntax beschreiben lassen, dann gilt:

$$\mathcal{L}_{LR(0)} = \mathcal{L}_{LR(k)} = \mathcal{L}_{det}{}^{7}.$$

10.2 LR-Parser

Ein LR-Parser besteht aus einer auf Grundlage der LR-Syntax G und des Lexikons generierten Parsingtabelle und einer einfachen Steuerprozedur zur Auswertung dieser Tabelle. Die Parsingtabelle besteht aus zwei Teilen: durch den ersten Teil wird die ACTION-Funktion definiert; durch den zweiten die GOTO-Funktion.

Die ACTION-Funktion nimmt einen Zustand s und ein terminales Symbol a_i[8] als Argumente und liefert einen der folgenden Werte:

shift neuer-zustand	
reduce regel	
accept	(Satz akzeptiert)
error	(Satz $\notin$ L(G))

Die Funktion GOTO nimmt einen Zustand s und ein nicht-terminales Symbol x als Argumente und liefert einen Zustand *s'* als Wert. Sie wird benötigt, um nach einer Reduce-Operation den neuen Zustand des Parsers zu bestimmen.

Eine Konfiguration eines LR-Parsers ist ein geordnetes Paar, das aus dem Stackinhalt und der noch nicht verarbeiteten Eingabe besteht:

$$< \overbrace{s_0\ X_1\ s_1\ X_2\ s_2\ \ldots\ X_m\ s_m}^{Stackinhalt},\ \overbrace{w_i\ w_{i+1}\ \ldots\ w_n\ \$}^{Rest\ der\ Eingabe}{}^{9} >$$

d.h. sie entspricht der Rechtssatzform $X_1 \ldots X_m w_i \ldots w_n$. Der Übergang von der aktuellen Konfiguration zur nächsten wird bestimmt durch ACTION(s_m, w_i)[10]:

1. Wenn ACTION(s_m, w_i) = *shift s*, dann hat die neue Konfiguration die Form:
   $$<s_0\ X_1\ s_1\ X_2\ s_2\ \ldots X_m\ s_m\ w_i\ s,\ w_{i+1}\ \ldots w_n\ \$>$$

[7] S. [?, S.143].

[8] Wir betrachten hier nur LR(k)-Syntaxen, mit $k \leq 1$. Für eine LR(k')-Syntax, mit $k' > 1$, ist das zweite Argument von ACTION eine Kette von Terminalsymbolen der Länge k'. Da in diesem Fall die Tabelle sehr stark wächst, ist diese Begrenzung sinnvoll.

[9] Das Symbol \$ ($\$ \notin v$) wird verwendet, um das Ende des Satzes zu markieren.

[10] s_m ist der aktuelle Zustand und w_i das nächste Symbol der noch nicht verarbeiteten Eingabe.

2. Wenn ACTION(s_m, w_i) = *reduce* $A \to \alpha$, dann hat die neue Konfiguration die Form: $<s_0\ X_1\ s_1\ X_2\ s_2\ \ldots X_{m-r}\ s_{m-r}\ A\ s,\ w_i\ w_{i+1}\ \ldots w_n\ \$>$, mit $s = GOTO(s_{m-r}, A)$ und $r = |\alpha|$; d.h. es werden zunächst 2*r Symbole vom Stack entfernt und anschließend die linke Regelseite der zum reduzieren verwendeten Regel und der mit GOTO berechnete neue Zustand auf den Stack geschoben.

3. Wenn ACTION(s_m, \$) = *accept*, dann endet die Analyse des Satzes erfolgreich.

4. Wenn ACTION(s_m, w_i) = *error*, dann bricht die Analyse des Satzes ab.

Die Prozedur, die das Verhalten eines LR-Parsers steuert, ist sehr einfach: Zu Beginn der Analyse eines Satzes $w = w_1 \ldots w_n$ befindet sich der Parser in der Konfiguration $<s_0, w_1 \ldots w_n\$>$; d.h. der Stack ist bis auf das Symbol für den Startzustand leer. Die nächste Konfiguration wird durch das oberste Symbol des Stacks und das erste Symbol des noch nicht verarbeiteten Teils des Satzes bestimmt. Es werden so lange neue Konfigurationen gebildet, bis entweder ACTION(s_i, \$) = *accept* bzw. ACTION(s_i, w_j) = *error*.

Beispiel (10-3)
Wenn die Syntax G_3 aus folgenden Regeln besteht

$1: E \to E + T$	$2: E \to T$	$3: T \to T * F$
$4: T \to F$	$5: F \to (E)$	$6: F \to id$

dann erhalten wir durch den im nächsten Abschnitt beschriebenen Algorithmus folgende Parsingtabelle:

Zustand	ACTION						GOTO		
	id	+	*	(	)	\$	E	T	F
0	s5			s4			1	2	3
1		s6				acc			
2		r2	s7		r2	r2			
3		r4	r4		r4	r4			
4	s5			s4			8	2	3
5		r6	r6		r6	r6			
6	s5			s4				9	3
7	s5			s4					10
8		s6			s11				
9		r1	s7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Für den Satz id * id + id erhalten wir folgenden Trace:

	Stack	Eingabe	Aktion
(1)	0	id * id + id $	*shift* 5
(2)	0 id 5	* id + id $	*reduce* 6
(3)	0 F 3	* id + id $	*reduce* 4
(4)	0 T 2	* id + id $	*shift* 7
(5)	0 T 2 * 7	id + id $	*shift* 5
(6)	0 T 2 * 7 id 5	+ id $	*reduce* 6

(7)	0 T 2 * 7 F 10	+ id $	*reduce* 3
(8)	0 T 2	+ id $	*reduce* 2
(9)	0 E 1	+ id $	*shift* 6
(10)	0 E 1 + 6	id $	*shift* 5
(11)	0 E 1 + 6 id 5	$	*reduce* 6
(12)	0 E 1 + 6 F 3	$	*reduce* 4
(13)	0 E 1 + 6 T 9	$	*reduce* 1
(14)	0 E 1	$	

10.3 Ein Algorithmus zur Generierung kanonischer LR(1)-Tabellen

Es gibt verschiedene Algorithmen zur Berechnung von Parsingtabellen für LR-Parser. Zu ihnen gehören u.a. der SLR-, der LLR- und der CLR-Algorithmus:[11]

1. Der SLR-Algorithmus („simple LR“) ist effizient und relativ einfach zu implementieren. Leider gibt es bestimmte LR-Syntaxen, für die dieser Algorithmus keine Parsingtabelle generieren kann.

2. Der LLR-Algorithmus („Look-ahead-LR“) erlaubt die Verarbeitung der Syntaxen der meisten Programmiersprachen und läßt sich mit gewissem Aufwand auch effizient implementieren.

3. Der CLR-Algorithmus („canonical LR“) erlaubt die Generierung von Parsingtabellen für beliebige LR-Syntaxen, ist allerdings auch deutlich komplexer als die beiden anderen Algorithmen.

Wegen seiner Leistungsfähigkeit beschreiben wir im folgenden den CLR-Algorithmus: Er berechnet zunächst eine Folge von LR(1)-Kantenmengen, aus der er dann in einem zweiten Schritt die Parsingtabelle generiert. Eine LR(1)-Kante repräsentieren wir als ein geordnetes Paar, das aus einer geteilten Produktion $A \to \alpha.\beta$ ($A \to \alpha\beta \in R$) und einem Look-ahead-Symbol $a \in V_T \cup \{\$\}$ besteht. Das Look-ahead-Symbol erlaubt es zu entscheiden, in welchen Situationen eine Regel für einen Reduktionsschritt verwendet werden darf:
Eine Regel $A \to \alpha$ darf nur dann zur Reduktion des Stacks verwendet werden, wenn die dem aktuellen Zustand des Parsers zugeordnete Kantenmenge die Kante $[A \to \alpha., a]$ enthält und das nächste Symbol der Eingabe a ist.

[11] Wir orientieren uns bei der Darstellung des Algorithmus eng an der in [?, S.198-248].

Zur Generierung der Folge von Kantenmengen verwenden wir die Prozedur BERECHNE-KANTENMENGEN, die ausgehend von der für die Regel S' $\rightarrow$ S gebildeten Kantenmenge alle weiteren Kantenmengen berechnet:

PROZEDUR BERECHNE-KANTENMENGEN

EINGABE: Eine Folge von LR(1)-Kantenmengen C.

METHODE:
Wiederhole bis C nicht weiter ergänzt werden kann:
Für jede Kanten-Menge I $\in$ C und jedes Symbol X $\in$ V, mit
GOTO-KANTEN(I, X) $\neq \emptyset$ und GOTO-KANTEN(I, X) $\notin$ C,
ergänze C um GOTO-KANTEN(I, X).

GOTO-KANTEN bildet ausgehend von einer Kantenmenge I für jedes X $\in V_N$, für das es mindestens eine Kante [A $\rightarrow \alpha$.Xβ, a] $\in$ I gibt, eine neue Kantenmenge J:

PROZEDUR GOTO-KANTEN

EINGABE: Eine Menge von LR(1)-Kanten I und ein Symbol X $\in$ V.
AUSGABE: Eine neue Menge von LR(1)-Kanten J.

METHODE:
J $\Leftarrow$ {[A $\rightarrow \alpha$X.β, a] | [A $\rightarrow \alpha$.Xβ, a] $\in$ I}.
RETURN(Closure$_{\text{ITEM}}$(J))

Die Prozedur CLOSURE$_{\text{ITEM}}$ nimmt eine Kantenmenge I als Argument und ergänzt sie durch Kanten, die mögliche Schritte im Analyseprozeß repräsentieren:

PROZEDUR CLOSURE$_{\text{ITEM}}$

EINGABE: Eine Menge von LR(1)-Kanten I.

METHODE:
Wiederhole bis keine neuen Kanten generiert werden können:
Für alle [A $\rightarrow \alpha$.Bβ, a] $\in$ I, B $\rightarrow \tau \in$ R und b $\in$ FIRST(βa):
Wenn <[B $\rightarrow$.τ, b] $\notin$ I>
Dann <I $\Leftarrow$ I $\cup$ {[B $\rightarrow$.τ, b]}>
RETURN(I)

Nach Berechnung der Folge von Kantenmengen wird die Parsingtabelle berechnet, wobei durch jede Kantenmenge ein Zustand des Parsers festgelegt wird. Die Berechnung der Tabelle gelingt nur, wenn für jedes Feld höchstens ein Wert spezifiziert wird; anderenfalls terminiert der Algorithmus mit einer Fehlermeldung.

ALGORITHMUS CLR(1)

EINGABE: Eine kontextfreie Syntax G = <V_N, V_T, S, R>, deren Regelmenge durch die Regel S' → S ergänzt wird.

AUSGABE: Eine Parsingtabelle für G / *False*.

METHODE:

Berechnung der Folge C = {I_0, I_1,..., I_n} von LR(1)-Mengen:

C ⇐ {$\text{Closure}_{\text{ITEM}}$({S' → .S, $})}
C ⇐ BERECHNE-KANTENMENGEN(C)

Berechnung der Parsingtabelle

a) ACTION-Felder:

Für i := 0 bis n:
Wenn <[A → α.aβ , b] ∈ I_i und GOTO-KANTEN(I_i, a) = I_j>
Dann <ACTION(i, a) = *shift j*>
Wenn <[A → α., a] ∈ I_i>
Dann <ACTION(i, a) = *reduce p*, mit p als Index der Regel A → α>
Wenn <[S' → S., $] ∈ I_i>
Dann <ACTION(i, $) = *accept*>

Wenn <ein Feld mehr als einen Eintrag enthält>
Dann <RETURN(*False*)>

b) GOTO-Felder:

Für alle I_i und X ∈ V_N gilt:
Wenn <GOTO-KANTEN(I_i, X) = I_j>
Dann <GOTO(i, X) = j>

Alle leeren Felder der Tabelle erhalten den Eintrag *error*.

Der Ausgangszustand des Parsers ist der Index der Kantenmenge, die aus dem Item [S' → .S, $] konstruiert wurde.

Beispiel (10-4)

Wenn G = <{S, C}, {c, d}, S, {r1:S → CC, r2:C → cC, r3:C → d}>, dann generiert CLR(1) folgende Kantenmengen:

CLOSURE({[S' → .S, $]}) =

I_0: S' → .S, $
S → .CC, $
C → .cC, c/d
C → .d, c/d

GOTO-KANTEN(I_0, S) =

I_1: S' → S., $

GOTO-KANTEN(I_0, C) =
I_2: S → C.C, \$
C → .cC, \$
C → .d, \$

GOTO-KANTEN(I_0, c) =
I_3: C → c.C, c/d
C → .cC, c/d
C → .d, c/d

GOTO-KANTEN(I_0, d) =
I_4: C → d., c/d

GOTO-KANTEN(I_2, C) =
I_5: S → CC., \$

GOTO-KANTEN(I_2, c)
I_6: C → c.C, \$
C → .cC, \$
C → .d, \$

=GOTO-KANTEN(I_2, c) =
I_7: C → d., \$

GOTO-KANTEN(I_3, C) =
I_8: C → cC., c/d

GOTO-KANTEN(I_6, C) =
I_9: C → cC., \$

Die Parsingtabelle hat dann folgende Form:

Zustand	ACTION			GOTO	
	c	d	\$	S	C
0	s3	s4		1	2
1			acc		
2	s6	s7			5
3	s3	s4			8
4	r3	r3			
5			r1		
6	s6	s7			9
7			r3		
8	r2	r2			
9			r2		

10.4 Effiziente Repräsentation von LR-Parsingtabellen

Die Größe der Parsingtabelle (Menge der Zustände und der in ihr verzeichneten Einträge) hängt von der Zahl der terminalen Symbole und Regeln der ihr zugrundeliegenden Syntax ab. Betrachtet man z.B. typische Programmiersprachen, so kann man von 50–100 terminalen Symbolen und etwa 100 Regeln ausgehen. Die auf Grundlage einer derartigen Syntax berechnete Parsingtabelle kann ohne weiteres aus mehreren hundert Zuständen und 20000 Einträgen allein im ACTION-Teil der Tabelle bestehen[12]. Ähnliche Dimensionen sind auch für Syntaxen natürlicher Sprachen zu erwarten, die mehr als nur einen trivialen Ausschnitt der Sprache beschreiben wollen.

[12] Vgl. [?, S.244-47].

Angesichts dieser Dimensionen ist es sinnvoll, sich um eine möglichst kompakte Repräsentation von Parsingtabellen zu bemühen. Wir werden im folgenden ein sehr einfaches Verfahren beschreiben, das es erlaubt, den ACTION-Teil einer Parsingtabelle relativ platzsparend zu repräsentieren, und das nur einen geringen Einfluß auf das Laufzeitverhalten des Parsers hat.: Für jeden Zustand wird eine Liste mit Symbol/Aktion-Paaren gebildet. Wenn *a* die Aktion ist, die in diesem Zustand am häufigsten auszuführen ist, dann werden alle Paare, in denen *a* vorkommt, durch das Paar <**any** *a*> ersetzt, das als letztes Element in die Liste eingefügt wird. Um die in einem Zustand auszuführende Aktion zu bestimmen, wird die diesem Zustand zugeordnete Liste von links nach rechts abgearbeitet, bis ein Eintrag gefunden ist, dessen Symbol mit dem nächsten Symbol der Eingabe übereinstimmt. Wird der **any**-Eintrag erreicht, dann wird die in ihm verzeichnete Aktion ausgeführt; d.h. **any** wird wie eine Variable behandelt, die mit beliebigen Symbolen *matcht*.

Beispiel (10-5)
Die Parsingtabelle aus Beispiel (10-4) kann wie folgt repräsentiert werden:

Zustand	ACTION
0	(<id s5>, <(s4>, <**any** *error*>)
1	(<+ s6>, <$ acc>, <**any** *error*>)
2	(<* s7>, <**any** r2>)
3	(<**any** r4>)
4	s. Zustand 1
5	(<**any** r6>)
6	s. Zustand 1
7	s. Zustand 1
8	(<+ s6>, <) s11>, <**any** *error*>)
9	(<* s7>, <**any** r1>)
10	(<**any** r3>)
11	(<**any** r5>)

Die durch die **any**-Einträge verursachte *Übergeneralisierung* führt dazu, daß der Parser bei fehlerhaften Eingaben später terminiert, als bei Verwendung einer nicht-komprimierten Parsingtabelle:

(A)	Stack	Eingabe	Aktion
(1)	0	id id * id $	*shift* 5
(2)	0 id 5	id * id $	*error*

(B)	Stack	Eingabe	Aktion
(1)	0	id id * id $	*shift* 5
(2)	0 id 5	id * id $	*reduce* 6
(3)	0 F 3	id * id $	*reduce* 4
(4)	0 T 2	id * id $	*reduce* 2
(5)	0 E 1	id * id $	*error*

Wie dieses Beispiel zeigt, terminiert der Parser bei Verwendung einer komprimierten Parsingtabelle (B) erst, sobald die nächste Shift-Aktion auszuführen ist.

10.5 Implementierung

Es folgen eine Lisp- und eine Prolog-Implementierung der in 10.2 beschriebenen Steuerprozedur für LR-Parser. Die Implementierung des Algorithmus CLR(1) zur Generierung kanonischer LR(1)-Tabellen überlassen wir dem Leser.

10.5.1 Lisp

Wie bei allen anderen Lisp-Programmen, die wir in den vergangenen Kapiteln entwickelt haben, setzen wir voraus, daß auf Syntax und Lexikon über die globalen Variablen *syntax* und *lexikon* zugegriffen werden kann. Die Parsingtabelle repräsentieren wir durch zwei komplexe Listen (je eine für den Action- und den Goto-Teil), die den globalen Variablen *action* und *goto* zugewiesen werden.

Deklarationen

```
(defvar *syntax*)          (defvar *lexikon*)
(defvar *action*)          (defvar *goto*)
```

Beispiel (10-6)
Die Parsingtabelle aus Beispiel (10-4) wird in folgender Form repräsentiert:

```
(setf *action*
  '((0 (c (shift 3)) (d (shift 4))) (1 ($ accept))
    (2 (c (shift 6)) (d (shift 7))) (3 (c (shift 3)) (d (shift 4)))
    (4 (c (reduce 3)) (d (reduce 3))) (5 ($ (reduce 1)))
    (6 (c (shift 6)) (d (shift 7))) (7 ($ (reduce 3)))
    (8 (c (reduce 2)) (d (reduce 2))) (9 ($ (reduce 2)))))

(setf *goto*
  '((0 (S 1) (C 2))
    (2 (C 5))
    (3 (C 8))
    (6 (C 9))))
```

Selektoren

```
;;; ACTION
;;; Selektorfunktion für den Action-Teil der Parsingtabelle.
(defmacro action (zustand symbol)
  `(second (assoc ,symbol (rest (assoc ,zustand *action*)))))
```

```
;;; GOTO
;;; Selektorfunktion für den Goto-Teil der Parsingtabelle.
(defmacro goto (zustand symbol)
  `(second (assoc ,symbol (rest (assoc ,zustand *goto*)))))

;;; KATEGORIE
;;; Liefert als Wert die lexikalische Kategorie des nächsten Wortes der Eingabe.
;;; Voraussetzung: jedem Wort wird genau eine Kategorie zugewiesen.
(defmacro kategorie (wort)
  `(if (eq ,wort '$)
       '$
       (first (second (assoc ,wort *lexikon*)))))
```

Hauptfunktionen

```
;;; LR-SHIFT
;;; Ausführen einer Shift-Operation.
(defun lr-shift (stack zustand symbol)
  (cons zustand (cons symbol stack)))

;;; LR-REDUCE
;;; Ausführen einer Reduce-Operation.
(defun lr-reduce (stack regel-index)
  (let* ((regel (nth (1- regel-index) (regeln)))
         (r-stack (nthcdr (* 2 (length (rest regel))) stack)))
    (cons (goto (first r-stack) (first regel)) (cons (first regel) r-stack))))

;;; RECOGNIZE_LR
;;; Hauptfunktion
(defun recognize_lr (satz)
  (let* ((stack '(0)) (symbol nil)
         (eingabe (append satz (list '$)))
         (wert nil))
    (loop
      (setq symbol (kategorie (first eingabe))
            wert   (action (first stack) symbol))
      (print stack)
      (cond ((not wert) (return nil))
            ((eq wert 'accept) (return t))
            ((eq (first wert) 'shift)
             (setq stack   (lr-shift stack (second wert) symbol)
                   eingabe (rest eingabe)))
            ((eq (first wert) 'reduce)
             (setq stack (lr-reduce stack (second wert)))))))))
```

10.5.2 Prolog

Repräsentation der Daten

Wir repräsentieren alle für das Parsen benötigten Informationen durch geeignete Fakten in der Wissensbasis:

1. Zur Repräsentation der Regeln wird das Prädikat regel/3 verwendet: *regel(Regelnummer, Linke_Seite, Rechte_Seite).*

2. Die Lexikoneinträge werden durch das lexkat/2-Prädikat spezifiziert: *lexkat(Wort, Kategorie).*

3. Zur Darstellung der Parsingtabelle verwenden wir die Prädikate action/3 bzw. goto/3: *action(Zustand, Symbol, Aktion)* bzw. *goto(Zustand, Symbol, Zustand1).*

Beispiel (10-7)

So erhalten wir ausgehend von der Syntax aus Beispiel (10-4) u.a.:

```
regel(1, s, [c, c]).        action(1, $, accept).
lexkat(d, c).               goto(2, c, 5).
```

Shift und Reduce

Bei jeder Shift-Operation werden eine lexikalische Kategorie und ein Zustandssymbol auf den Stack geschoben. Bei Aufruf von shift/4 sind die ersten drei Argumente mit dem alten Stack, dem neuen Zustand und der Kategorie instantiiert. Das vierte Argument liefert den neuen Stack:

```
% shift/4
shift(Stack, Zustand, Kategorie, [Zustand, Kategorie | Stack]).
```

Bei einer Reduce-Operation sind n*2 Symbole aus dem Stack zu entfernen, mit n als der Länge der rechten Seite der zur Reduktion verwendeten Regel. Die ersten beiden Argumente von reduce/3 bilden die Regel und der ursprüngliche Stack; das dritte Argument wird mit dem reduzierten Stack instantiiert:

```
% reduce/3
reduce(Regel, Stack, [Zustand1, Links | [Zustand | Rest]]) :-
    regel(Regel, Links, Rechts),
    reduce1(Rechts, Stack, [Zustand | Rest]),
    goto(Zustand, Links, Zustand1).

% reduce1/3
% Dieses Prädikat entfernt für jedes Symbol der rechten Regelseite zwei Symbole
% (ein Zustands- und ein Kategoriensymbol) aus dem Stack:
```

```
reduce1([ ], Stack, Stack).
reduce1([Symbol | Symbole], [Zustand, Knoten | Rest], R) :-
    reduce1(Symbole, Rest, R).
```

Recognize

```
% recognize/1
% Der Satz wird mit dem Satzbegrenzungszeichen ergänzt und die Analyse gestartet.

recognize(Satz) :-
    append(Satz, [$], Satz1),
    recognize([0], Satz1).

% recognize/2
% Das Prädikat hat als Argumente den Stack (oberste Symbol = erstes Symbol der
% Liste) und den noch zu analysiereden Teil der Eingabe.

% Analyse terminiert erfolgreich
recognize([Zustand | Rest], [Wort | Worte]) :-
    lexkat(Wort, Kategorie),
    action(Zustand, Kategorie, accept).
% Analyse ist nach einer Shift-Operation fortzusetzen
recognize([Zustand | Rest], [Wort | Worte]) :-
    lexkat(Wort, Kategorie),
    action(Zustand, Kategorie, shift(Zustand1)),
    shift([Zustand | Rest], Zustand1, Kategorie, Stack),
    recognize(Stack, Worte).
% Analyse ist nach einer Reduce-Operation fortzusetzen
recognize([Zustand | Rest], [Wort | Worte]) :-
    lexkat(Wort, Kategorie),
    action(Zustand, Kategorie, reduce(Regel)),
    reduce(Regel, [Zustand | Rest], [Zustand1 | Rest1]),
    recognize([Zustand1 | Rest1], [Wort | Worte]).
```

Aufgaben

10.1 Implementieren Sie den CLR(1)-Algorithmus so, daß er

(a) mit einer Fehlermeldung terminiert, sobald es zu Eintragskonflikten kommt;

(b) auch Felder mit Mehrfacheinträgen generiert und

(c) eine komprimierte Parsingtabelle (**any**-Eintrag) erzeugt.

10.2 Erweitern Sie den LR-Recognizer so, daß er Tabellen mit Mehrfacheinträgen durch Verwendung von Backtracking verarbeiten kann.

10.3 Schreiben Sie ein Programm, das für ein beliebiges $k \geq 0$ prüft, ob es sich bei einer kontextfreien Syntax G um eine LR(k)-Syntax handelt.

Kapitel 11

Der Tomita-Algorithmus

Tomitas Algorithmus ist eine Weiterentwicklung des LR-Parsingalgorithmus. Es ist ein genereller Parsingalgorithmus, d.h. er kann für beliebige kontextfreie Sprachen verwendet werden. Der wichtigste Unterschied zu dem einfachen LR-Algorithmus, den wir in Kapitel 10 beschrieben haben, liegt in der Verwendung eines *graph-strukturierten* Stacks, der eine effiziente Verarbeitung von Parsingtabellen mit Mehrfacheinträgen erlaubt.

11.1 LR-Parsing beliebiger kontextfreier Syntaxen

Wird das im letzten Kapitel beschriebene Konstruktionsverfahren für Parsingtabellen auf beliebige kontextfreie Syntaxen angewendet, dann führt es in den Fällen, in denen es sich bei der Syntax nicht um eine LR-Syntax handelt, dazu, daß die Parsingtabelle in einem oder mehreren Feldern Mehrfacheinträge aufweist. Solche Tabellen lassen sich nicht innerhalb eines einfachen LR-Parsers verwenden, da sich nicht mehr in jedem Zustand die nächste Operation eindeutig bestimmen läßt; d.h. der Parser kann nicht mehr deterministisch operieren[1].

Wenn sich z.B. ein mit der Tabelle aus dem folgenden Beispiel operierender LR-Parser im Zustand 11 bzw. 12 befindet und das nächste zu verarbeitende Wort eine Präposition ist, können zwei Operationen ausgeführt werden:

- eine *Shift*-Operation, die den Parser in den Zustand 6 übergehen läßt oder
- eine *Reduce*-Operation unter Verwendung der Regel 6[2].

Beispiel (11-1)

[1] Aus diesem Grund brechen die Konstruktionsverfahren normalerweise ab, sobald einem Feld verschiedene Werte zugewiesen werden.

[2] Die Ambiguität der Syntax liegt in der Behandlung satzfinaler Präpositionalphrasen bzw. mehrerer aufeinanderfolgender Präpositionalphrasen.

Wenn G = <V_N, V_T, S, R> eine kontextfreie Syntax ist, die die folgenden Regeln enthält:

1 : S → NP VP	2 : S → S PP.	3 : NP → n
4 : NP → det n	5 : NP → NP PP	6 : PP → prep NP
7 : VP → v NP		

dann erhalten wir für G (wenn wir dafür sorgen, daß der Algorithmus nicht abbricht, sobald er auf Mehrfacheinträge stößt) folgende Parsingtabelle:

Zustand	ACTION					GOTO			
	det	n	v	prep	$	NP	PP	VP	S
0	s3	s4				2			1
1				s6	acc		5		
2			s7	s6			9	8	
3		s10							
4			r3	r3	r3				
5				r2	r2				
6	s3	s4				11			
7	s3	s4				12			
8				r1	r1				
9			r5	r5	r5				
10			r4	r4	r4				
11			r6	r6,s6	r6		9		
12				r7,s6	r7		9		

Wir werden jetzt drei Strategien vorstellen, die eine zunehmend effizientere Verarbeitung von Parsingtabellen mit Mehrfacheinträgen erlauben. Gemeinsam ist ihnen, daß sie eine komplexere Form der Informationsverwaltung voraussetzen als einfachere LR-Parser. Wir bezeichnen im folgenden die zur Informationsspeicherung verwendete Datenstruktur als **Stackgruppe**. Die drei Strategien unterscheiden sich durch die in der Stackgruppe gespeicherten Objekte und den auf ihnen definierten Operationen.

11.1.1 Stack-Listen

Die einfachste der drei Strategien beschränkt sich darauf, alternative Analysemöglichkeiten eines Satzes, wie sie durch die Mehrfacheinträge in einer Parsingtabelle repräsentiert werden, durch separate Prozesse verfolgen zu lassen:

Zu Beginn der Analyse gibt es zunächst genau einen Prozeß P, und die Stackgruppe enthält einen Stack S. Sobald P auf ein Feld stößt, das mehr als einen Eintrag enthält, wird P in verschiedene Prozesse aufgespalten, so daß jede Analysemöglichkeit durch einen eigenen Prozeß verfolgt werden kann. Jedem dieser Prozesse ist ein eigener Stack - eine Kopie von S - zugeordnet; d.h. es muß nicht ein Stack, sondern eine *Liste* von Stacks verwaltet werden. Dieser Vorgang wiederholt sich jedesmal, sobald einer der Prozesse auf ein Feld mit Mehrfacheinträgen stößt. Wenn ein Prozeß nicht fortgesetzt werden kann; d.h. er auf einen *error*-Eintrag in der Tabelle stößt, terminiert er, und sein Stack wird aus der Stack-Liste entfernt.

Jeder der aktiven Prozesse korrespondiert mit einem einfachen LR-Parser. Wir setzen voraus, daß sie **synchron** arbeiten: Wenn ein Prozeß durch eine Shift-Aktion das nächste Symbol der Eingabe auf dem Stack ablegt, wird er suspendiert, bis alle anderen Prozesse dieses Symbol ebenfalls in ihren Stack übertragen haben.

Diese Strategie mit ihrer Verwendung 'paralleler' Prozesse (Pseudo-Parallelität), kann als Realisierung der Breadth-first-Strategie aufgefaßt werden. Sie ist durch drei Grundoperationen charakterisiert: *Initialisieren* der Stackgruppe, *Kopieren* eines Stacks und *Löschen* eines Stacks. Ihr entscheidender Nachteil liegt darin, daß es keine Kommunikation zwischen den Prozessen gibt. Da ein Prozeß nicht auf die Ergebnisse eines anderen zugreifen kann, erzeugt die so erforderliche Duplizierung von Informationen (Stackinhalten) ein hohes Maß an Redundanz. Mit dem Auftreten von Ambiguitäten wächst die Zahl der Stacks exponentiell.

Beispiel (11-2)
Die Syntax und Parsingtabelle aus Beispiel (11-1) vorausgesetzt, erhalten wir für den Satz „Die Verwaltung sucht eine Nadel im Heuhaufen.“ folgendes Ergebnis:

Stackgruppe

Stack S_1	
0	
0–det–3	
0–det–3–n–10	
0–np–2	(Subjekt-NP gefunden)
0–np–2–v–7–det–3–n–10	
0–np–2–v–7–np–12	(Objekt-NP gefunden)

ACTION(12, prep*) enthält zwei Einträge, also wird eine Kopie von S_1 angefertigt:

Stackgruppe

Stack S_1	Stack S_2
0–np–2–v–7–np–12	**0–np–2–v–7–np–12**
0–np–2–vp–8	
0–s–1	
0–s–1–**prep–6**	0–np–2–v–7–np–12–**prep–6**
0–s–1–**prep–6–n–4**	0–np–2–v–7–np–12–**prep–6–n–4**
0–s–1–**prep–6–np–11**	0–np–2–v–7–np–12–**prep–6–np–11**
0–s–1–pp–5	0–np–2–v–7–np–12–pp–9
0–s–1	0–np–2–v–7–np–12
accept	0–np–2–vp–8
	0–s–1
	accept

Schon dieses kleine Beispiel läßt gut erkennen, daß verschiedene Prozesse z.T. dieselben Aktionen ausführen.

11.1.2 Baum-strukturierter Stack

Die Synchronisierung der Prozesse legt die Verwendung einer effizienteren Informationsverwaltung nahe: Wenn zwei synchronisierte Prozesse sich in demselben Zustand befinden, d.h. wenn die Zustandssymbole, die in ihren Stacks oben liegen, denselben Zustand bezeichnen, dann führen sie so lange die gleichen Operationen aus, bis eine *Reduce*-Aktion dieses Zustandssymbol vom Stack entfernt. Um eine unnötige Protokollierung identischer Aktionssequenzen in verschiedenen Stacks zu vermeiden, kann man Prozesse, die sich im gleichen Zustand befinden, durch Kombination ihrer Stacks zu einem Prozeß zusammenfassen, der erst wieder in die ursprünglichen Prozesse aufgespalten wird, nachdem eine *Reduce*-Aktion es erforderlich macht.

Der Stack des neuen Prozesses kann als Baum repräsentiert werden, dessen Wurzel den Zustand der beiden zusammengefaßten Prozesse repräsentiert. Die Stackgruppe besteht jetzt aus einer Folge von baum-strukturierten Stacks; denn nicht immer lassen sich alle Prozesse an einem Punkt verknüpfen. So können z.B. drei Prozesse P_{j1}, P_{j2}, P_{j3} zu einem Prozeß P_J und zwei weitere Prozesse P_{k1}, P_{k2} zu einem anderen Prozeß P_K kombiniert werden. Die Stackgruppe enthält in diesem Fall zwei baum-strukturierte Stacks. Obwohl dieses Verfahren erheblich effizienter ist, als das vorher beschriebene, wird immer noch beim Aufspalten eines Prozesses der Stackinhalt vollständig kopiert. Die Zahl der zu bildenden Prozesse (repräsentiert durch die Äste der baum-strukturierten Stacks) wächst weiterhin exponentiell.

Beispiel (11-3)

Wir verwenden wieder die Syntax und Parsingtabelle aus Beispiel (11-1). Zunächst verläuft die Analyse wie in Beispiel (11-2) beschrieben:

$$\left[\begin{array}{c} \vdots \\ \text{0–np–2–v–7–np–12} \end{array}\right]$$

Der Mehrfacheintrag in ACTION(12, prep*) führt zur Teilung des Stacks:

$$\left[\begin{array}{l|l} \text{0–np–2–v–7–np–12} & \text{0–np–2–v–7–np–12} \\ \text{0–np–2–vp–8} & \\ \text{0–s–1} & \\ \text{0–s–1–prep–}\mathbf{6} & \text{0–np–2–v–7–np–12–prep–}\mathbf{6} \end{array}\right]$$

Beide Prozesse befinden sich jetzt im selben Zustand, also werden sie kombiniert:

$$\left[\begin{array}{l} \text{6–n–4} \\ \text{6–np–11} \end{array}\right]$$

Die *reduce*-Aktionen machen es erforderlich, beide Prozesse separat zu verwalten:

$$\left[\begin{array}{l|l} \text{0–s–1–pp–5} & \text{0–np–2–v–7–np–12–pp–9} \\ \text{0–s–1} & \text{0–np–2–v–7–np–12} \\ \textit{accept} & \text{0–np–2–vp–8} \\ & \text{0–s–1} \\ & \textit{accept} \end{array}\right]$$

11.1.3 Graph-strukturierter Stack

Charakteristisch für die beiden oben beschriebenen Strategien ist, daß bei der Aufspaltung eines Prozesses sein Stack komplett kopiert wird und es so durchaus möglich ist, daß die untersten (linken) Symbole der Stacks von Prozessen, die zuletzt ganz verschiedene Aktionssequenzen durchlaufen haben, trotzdem identisch sind. Diese unnötige Duplikation von Informationen läßt sich vermeiden, wenn nach Aufspalten eines Prozesses der Stack nicht einfach kopiert, sondern als ein Baum repräsentiert wird, dessen Spitze den Anfangszustand des ursprünglichen Prozesses repräsentiert und dessen Äste den unterschiedlichen Verlauf der neuen Prozesse abbilden.

Wenn die Stacks von Prozessen, die sich im gleichen Zustand befinden, wie zuvor kombiniert werden, dann kann der Stack als ein gerichteter azyklischer Graph repräsentiert werden. Die Stackgruppe enthält also nicht mehr eine Liste von Stacks, sondern nur noch einen, unter Umständen allerdings recht komplexen, *graph-strukturierten* Stack[3].

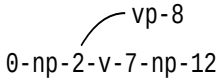
Beispiel (11-4)

Wir gehen wieder von Syntax, Parsingtabelle und Satz aus Beispiel (11-1) aus:

(1) `0-np-2-v-7-np-12`

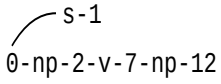
Inhalt des Stacks nach Verarbeitung der Objekt-NP.

(2)



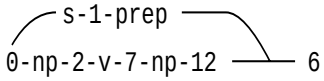
Der Doppeleintrag in der Parsingtabelle erlaubt es, entweder ein Symbol einzulesen oder den Stack zu reduzieren. Die Reduce-Operation hat Vorrang.

(3)



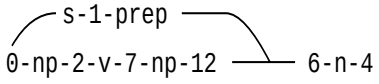
Der Stack wird erneut reduziert. Jetzt ist eine Situation erreicht, in der der Stack nicht weiter reduziert werden kann.

(4)

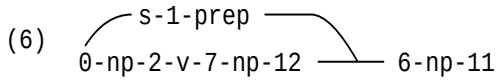


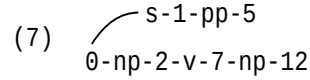
Die Shift-Operation erlaubt es, beide Prozesse zu kombinieren.

(5)



[3]Wie Tomita selbst bemerkt, ähnelt der graph-strukturierte Stack stark der Chart eines Chart-Parsers. Wie eine Chart erlaubt der Stack die Speicherung von partiellen Analyseresultaten.





Die Reduce-Operation erzwingt eine erneute Aufspaltung in zwei Prozesse.

11.1.4 Effiziente Strukturverwaltung

In der Regel wird von Parsern für natürliche Sprachen erwartet, daß sie alle Strukturen für einen Satz generieren, die sich relativ zur zugrundeliegenden Syntax für ihn bilden lassen. Da der Ambiguitätsgrad[4] von Sätzen exponentiell mit ihrer Länge wächst, würde normalerweise auch die Zeit, die ein effizienter Parser benötigt, um alle Strukturbeschreibungen eines Satzes auszugeben, exponentiell wachsen. Neben einem effizienten Analyseverfahren für Sätze ist daher auch eine effiziente Repräsentation für die für sie generierten Strukturen notwendig. Zu diesem Zweck kombiniert Tomita zwei schon früher in anderen Systemen verwendete Techniken: das „*sub-tree sharing*“ und das „*local ambiguity packing*“.

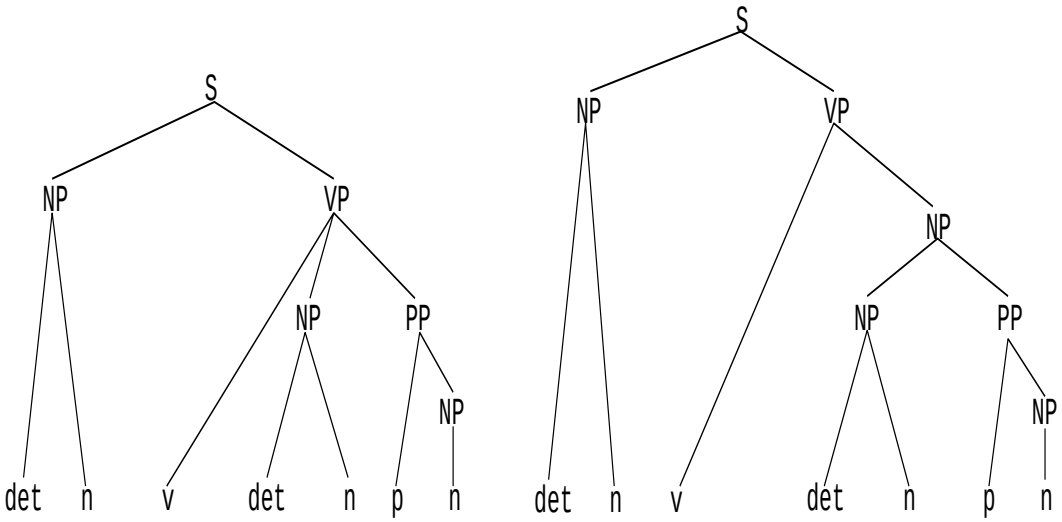
sub-tree sharing. Wenn verschiedene Strukturbeschreibungen eines Satzes identische Teilstrukturen für denselben Satzabschnitt enthalten, dann ist es vollkommen ausreichend, diese Teilstrukturen nur einmal zu repäsentieren. Das Ergebnis ist eine „shared forest“-Repräsentation: Alle Strukturbeschreibungen, die gemeinsame Teilstrukturen aufweisen, werden *aufeinandergelegt* und es entsteht so ein Graph, der die gemeinsamen Teilstrukturen nur einmal repräsentiert.

Local Ambiguity Packing. Verschiedene Strukturbeschreibungen eines Satzes repräsentieren eine *lokale Ambiguität* des Satzes, wenn sie Teilstrukturen für denselben Satzabschnitt enthalten, die zwar nicht isomorph sind, aber gemeinsame Terminalknoten besitzen und die Etiketten ihrer Wurzelknoten gleich sind. Um zu vermeiden, daß, wenn ein Satz viele solcher lokaler Ambiguitäten aufweist, sein Ambiguitätsgrad exponentiell wächst, werden zu ihrer Repräsentation *gepackte Knoten* („packed nodes“) verwendet: Die Wurzelknoten der eine lokale Ambiguität repräsentierenden Teilbäume werden zu einem (gepackten) Knoten zusammengefaßt, der von übergeordneten Strukturen als einfacher Knoten behandelt wird. Die zu einem gepackten Knoten zusammengefaßten Knoten werden als „*Subknoten*“ dieses gepackten Knotens bezeichnet.

Durch Kombination dieses Verfahrens mit dem zuvor beschriebenen werden Strukturbeschreibungen eines Satzes, soweit möglich, miteinander verknüpft und als „packed shared forests“ repräsentiert.

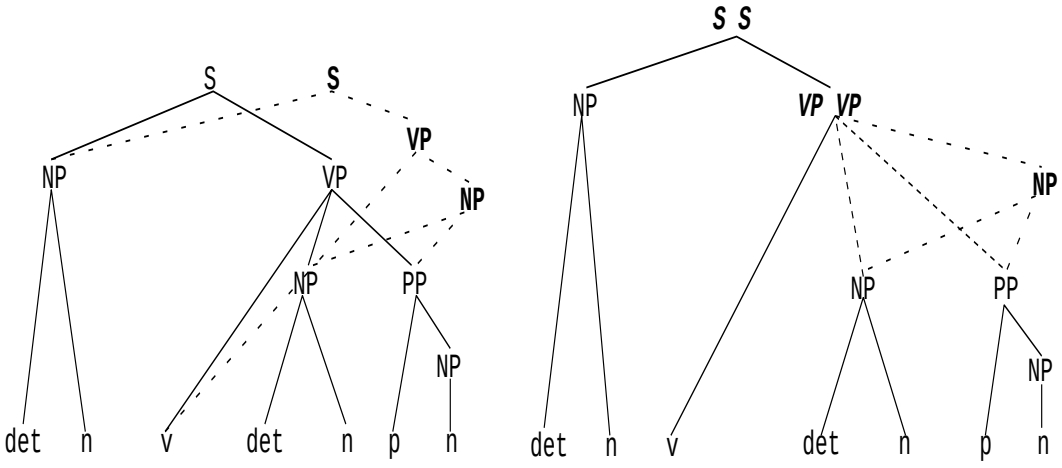
[4] Als *Ambiguitätsgrad* eines Satzes wird die Zahl der Strukturen bezeichnet, die ihm durch die Syntax zugeordnet werden können.

Beispiel (11-5)
Betrachten wir den Fall, daß ein Parser für einen Satz die folgenden beiden Strukturbeschreibungen generiert:



Dann gilt:

- Die Nominalphrasen und die Präpositionalphrase haben dieselbe Struktur.
- Die S-Knoten und die VP-Knoten lassen sich zusammenfassen.



subtree-sharing *subtree-sharing & local ambiguity packing*

11.2 Der Algorithmus

Wir entwickeln Tomitas Algorithmus in zwei Schritten: Zunächst formulieren wir ihn als Erkennungsalgorithmus für beliebige kontextfreie Syntaxen[5] (inklusive Tilgungsregeln) und erweitern ihn dann zu einem Parsingalgorithmus, der *gepackte* Strukturbeschreibungen generiert[6].

11.2.1 Der Erkennungsalgorithmus

Abbildung (11-1) gibt einen Überblick über die Prozeduren, die wir im folgenden definieren werden:

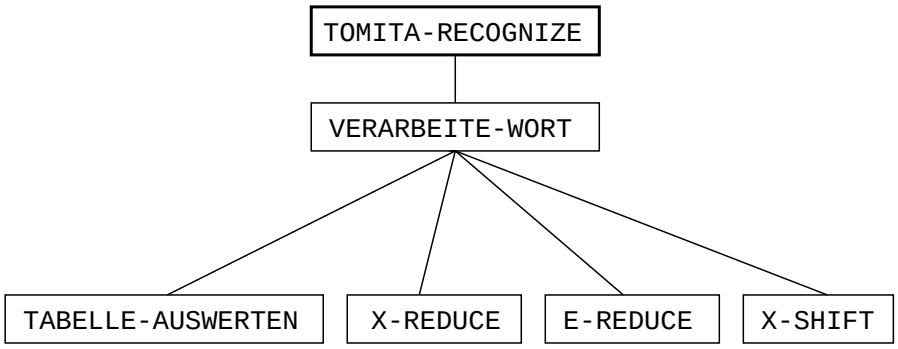


Abbildung (11-1)

Die Prozedur VERARBEITE-WORT steuert die Verarbeitung des Satzes: Sie stellt sicher, daß alle Analysemöglichkeiten verfolgt werden und daß das nächste Wort erst dann in den Stack übertragen wird (X-SHIFT), wenn keine Reduce-Operation mehr ausgeführt werden kann. TABELLE-AUSWERTEN bestimmt anhand eines Eintrags in der Parsingtabelle die Menge der Aktionen, die im nächsten Schritt ausgeführt werden können; die Prozeduren X-REDUCE, E-REDUCE und X-SHIFT steuern die Ausführung von Reduce-Operationen (ohne bzw. mit Tilgungsregeln) und Shift-Operationen.

Der Stack, mit dem der Tomita-Parser arbeitet, läßt sich als ein gerichteter azyklischer Graph (*directed acyclic graph* - DAG) repräsentieren, der nach seiner Initialisierung aus dem Terminalknoten v_0 besteht, der das Stackende (*bottom*) repräsentiert. Die Knoten des Stacks sind etikettiert und repräsentieren entweder einen Zustand des Parsers z oder ein auf den Stack geschobenes Symbol $s \in (V_N \cup V_T)$. Es gibt also zwei Typen von Knoten: *Zustandsknoten* und *Symbolknoten*. Da sich in einem Pfad im Stack Γ Zustands- und Symbolknoten abwechseln, gilt: Wenn v_i ein Knoten vom Typ **X** ist, dann ist jeder Knoten, zu dem von v_i ein Pfad der Länge 2 *

[5] Einzige Ausnahme bilden unbegrenzt ambige bzw. zyklische Syntaxen; vgl. [?, S.72/73].
[6] Wir orientieren uns eng an Tomitas eigener Darstellung; vgl. [?, S.49-69].

n führt, auch ein Knoten vom Typ **X**. Wenn v_i ein Knoten mit dem Etikett s_i ist, dann notieren wir das als: „$s_i(v_i)$“.

Wenn G eine kontextfreie Syntax G ist, die Tilgungsregeln enthält, und $w = w_1 \ldots w_n$ ($n \geq 0$) ein beliebiger Satz aus L(G) ist, dann können bei der Analyse von w vor w_1 und nach w_n und zwischen allen w_i, w_{i+1} ($1 \leq i < n$) eine beliebige, endliche Zahl von e Symbolen erzeugt werden; d.h. w kann notiert werden als:

$$e_{01} \ldots e_{0j}\ w_1\ e_{11} \ldots e_{1k}\ w_2 \ldots\ldots w_n\ e_{n1} \ldots e_{nl}\ (j, k, l \geq 0).$$

Um eine transparente Formulierung des Algorithmus zu erhalten, werden die Zustandsknoten von Γ in einer Folge

$$\mathbf{U} = U_{00}, \ldots, U_{0j}, \ldots, U_{n0}, \ldots, U_{nl}\ (j,l \geq 0)$$

von Mengen zusammengefaßt, wobei U_{i0} die Menge der Zustandsknoten ist, die nach Verarbeitung des i-ten Symbols der Eingabe erzeugt wurden, und U_{ij} ($j > 0$) ist die Menge der Zustandsknoten, die nach Verarbeitung des j-ten e, das auf das i-te Eingabesymbol folgt, generiert wurden.

Für die Darstellung der Prozeduren werden folgende Variablen und Funktionen verwendet:

Variablen

- Γ - Der graph-strukturierte Stack.
- $U_{i,j}$ - Eine Menge von Zustandsknoten (s.o.).
- A - Die Menge der momentan *aktiven Knoten*; d.h. die Knoten, die am Anfang eines Pfades liegen, der durch eine Reduce- oder Shift-Operation modifiziert werden kann.
- Q - Eine Menge von Paaren <v, s>, die aus einem aktiven (Zustands-) Knoten v und einem Zustand s bestehen. v ist der Anfangsknoten eines Pfades, der durch eine Shift-Operation zu erweitern ist, und s der Zustand des Parsers nach dieser Operation.
- R - Eine Menge von Tripeln <v, x, p>, die aus einem Zustandsknoten v, einem Symbolknoten x, mit $x \in NACHFOLGER(v)$, und einem Regelindex p bestehen. Sie verweisen auf einen Pfad, der mit der Kante <v, x> beginnt und der durch Anwendung der Regel p zu reduzieren ist.
- R_e - Eine Menge von Paaren <v, p>, die aus einem aktiven Zustandsknoten v und dem Index einer Tilgungsregel p bestehen und die auf den Startknoten eines Pfades verweisen, der durch die Regel p zu reduzieren ist.

Funktionen

(a) für die Parsingtabelle P:

- *GOTO(Zustand, Symbol)* liefert als Wert den Inhalt des so bezeichneten Feldes der GOTO-Tabelle von P.

- *ACTION(Zustand, Symbol)* liefert den Inhalt des so bezeichneten Feldes der ACTION-Tabelle von P.

(b) für den Stack:

- *NACHFOLGER(v)* nimmt einen Knoten von Γ als Argument und liefert als Wert die Menge aller Knoten in Γ, zu denen von v eine Kante führt.
- *ZUSTAND(v)* nimmt einen Zustandsknoten von Γ als Argument und liefert sein Etikett.
- *SYMBOL(v)* nimmt einen Symbolknoten von Γ als Argument und liefert sein Etikett.

(c) für Regeln aus G:

- *LINKS(p)* liefert für eine Regel p die linke Seite von p als Wert.
- *RECHTS(p)* liefert für eine Regel p die rechte Seite von p als Wert.
- $|p|$ bezeichnet die Länge (d.h. Zahl der Symbole) der Regel p.

TOMITA-RECOGNIZE. Der graph-strukturierte Stack wird initialisiert. Er besteht zunächst nur aus dem Knoten, der den Startzustand des Parsers repräsentiert (s_0 bezeichnet diesen Startzustand). $U_{0,0}$ ist die Menge der Knoten, die vor Beginn der Analyse des Satzes gebildet werden. Die Analyse des Satzes wird durch die Prozedur VERARBEITE-WORT gesteuert.

ALGORITHMUS TOMITA-RECOGNIZE

DATEN: Eine kontextfreie Syntax G = <V_N, V_T, S, R> und eine Parsingtabelle P_G.
EINGABE: Ein Satz w = $w_1 \ldots w_n$ ($n \geq 0$).
AUSGABE: *True*/*False*.
ARBEITSSTRUKTUREN:
Γ - Ein graph-strukturierter Stack.
Anfangswert: leer
$U_{0,0}$ - Eine Menge von Knoten aus Γ.
Anfangswert: leer

METHODE:
$w_{n+1} \Leftarrow \$$.
$\Gamma \Leftarrow s_0(v_0)$.
RESULTAT $\Leftarrow$ *False*.
$U_{0,0} \Leftarrow \{v_0\}$.
Für i = 0 bis n:
<*Verarbeite-Wort*(i)>
<RETURN(RESULTAT)>

VERARBEITE-WORT. Zunächst werden die globalen Variablen A, R, R_e und Q initialisiert: Die Menge der aktiven Knoten besteht aus der Menge aller durch die im letzten Schritt ausgeführten Shift-Operationen[7] gebildeten Knoten (bzw. für i = 0 aus dem den Startzustand repräsentierenden Knoten). R, R_e und Q sind leer.

PROZEDUR VERARBEITE-WORT
(* Verarbeitet das nächste Wort *)

EINGABE: Eine Zahl i, die auf das im letzten Schritt verarbeitete Wort des Satzes verweist.

METHODE:

```
A ⇐ U_{i,0}.
R, R_e, Q ⇐ ∅.
Wiederhole bis: A = ∅ ∧ R = ∅ ∧ R_e = ∅
   Wenn   <A ≠ ∅>
   Dann   <Tabelle-Auswerten(A)>
   Sonst
          Wenn   <R ≠ ∅>
          Dann   <X-Reduce(R)>
          Sonst
                 Wenn   <R_e ≠ ∅>
                 Dann   <E-Reduce(R_e)>
<X-Shift(Q)>
```

TABELLE-AUSWERTEN. Diese Prozedur nimmt eine Menge von aktiven (Zustands-) Knoten als Argument und wertet für jeden durch sie repräsentierten Zustand z und das nächste Wort der Eingabe w_{i+1} den Eintrag in der Parsingtabelle (ACTION(z, w_{i+1})) aus: Sie bereitet Reduce- bzw. Shift-Aktionen vor und kann den Wert der booleschen Variable RESULTAT verändern.

Die Prozedur TABELLE-AUSWERTEN sorgt dafür, daß für jeden aktiven Knoten alle Einträge des relevanten Feldes der Parsingtabelle ausgewertet werden; d.h. es werden in R, R_e und Q die für die Ausführung der Operationen notwendigen Informationen gespeichert. Anschließend werden alle Reduce-Operationen (ohne/mit Tilgungsregeln) ausgeführt, was zu neuen aktiven Knoten führen kann. Erst wenn keine Reduce-Operationen mehr ausgeführt werden können, werden die in dieser Situation möglichen Shift-Operationen ausgeführt.

[7] Da mehrere Knoten aktiv sein können, ist es auch möglich, daß mehr als eine Shift-Operation ausgeführt wird.

PROZEDUR TABELLE-AUSWERTEN
(* Berechnen der nächsten Operationen *)

EINGABE: Eine Menge von aktiven Knoten A.

METHODE:
Entferne ein Knoten v ∈ A.
Für alle $\alpha \in Action(Zustand(\mathrm{v}), \mathrm{w_{i+1}})$:
Wenn <α = *accept*>
Dann <RESULTAT ⇐ *True*>
Wenn <α = *shift s*>
Dann <Q ⇐ Q ∪ {<v, s>}>
Wenn <α = *reduce p*>
Dann
Wenn <p ist keine *e*-Regel>
Dann
Für alle x, mit x ∈ *Nachfolger*(v):
<R ⇐ R ∪ {<v, x, p>}>
Sonst <$\mathrm{R_e}$ ⇐ $\mathrm{R_e}$ ∪ {<v, p>}>

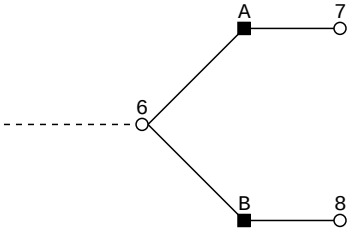
X-SHIFT. Für jeden Zustand, in den der Parser durch Ausführung einer Shift-Operation übergehen kann, wird ein Knoten in Γ gebildet, von dem aus eine Kante zu einem das nächste Wort der Eingabe ($\mathrm{w_{i+1}}$) repräsentierenden Knoten führt. Dieser Knoten wiederum wird durch Kanten mit allen Knoten verbunden, die den Zustand des Parsers vor dieser Shift-Operation repräsentieren.

PROZEDUR X-SHIFT
(* Durchführen aller zulässigen Shift-Operationen *)

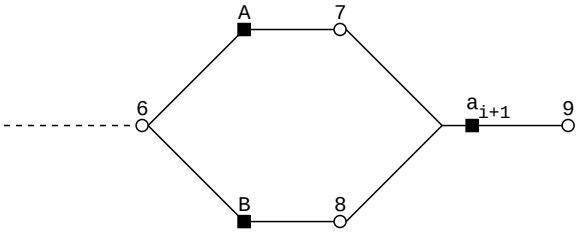
EINGABE: Eine Menge Q von Paaren <v, z>.

METHODE:
$\mathrm{U_{i+1,0}}$ ⇐ ∅.
Für alle z, mit <v, z> ∈ Q:
<Erzeuge zwei Knoten z(k) und $\mathrm{w_{i+1}}$(k') in Γ.>
<Erzeuge eine Kante von k nach k' in Γ.>
<$\mathrm{U_{i+1,0}}$ ⇐ $\mathrm{U_{i+1,0}}$ ∪ {k}.>
Für alle v, mit <v, z> ∈ Q:
<Erzeuge eine Kante von k' nach v in Γ.>

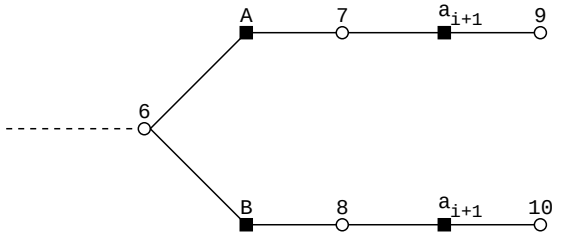
Beispiel (11-6)
Wenn Q = {< 7, 9>, < 8, 9>} und der graph-strukturierte Stack Γ die folgende Teilstruktur enthält:



dann erhalten wir nach Ausführung der Shift-Operationen die Struktur:



Wenn aber gilt: Q = {< 7, 9>, < 8, 10>}, dann ergibt sich die folgende Struktur:



X-REDUCE. Bei der Ausführung einer Reduce-Operation in einem einfachen LR-Parser werden zuerst 2 * n Symbole aus dem Stack entfernt, mit n als der Länge der rechten Regelseite der zum Reduzieren verwendeten Regel p. Anschließend werden das Symbol der linken Regelseite von p und der aus dem GOTO-Teil der Parsingtabelle berechnete neue Zustand auf den Stack geschoben.

Bei Verwendung eines graph-strukturierten Stacks ist eine etwas komplexere Reduce-Operation erforderlich: Wenn s ein Wurzelknoten von Γ ist, der einen Zustand des Parsers repräsentiert, in dem eine Reduce-Operation mit einer Regel p durchzuführen ist, dann sei Z die Menge aller Zustandsknoten, zu denen von s ein Pfad der Länge 2 * $|RECHTS(\mathrm{p})|$ führt.

Z_i sei eine Teilmenge von Z, die alle Knoten aus Z zusammenfaßt, die einen gemeinsamen Vorgänger (letzter bei der Reduktion zu entfernender Symbolknoten) besitzen, und von denen aus nach der Reduktion derselbe Zustand z erreicht wird. Für jedes derartige $Z_i \subseteq Z$ werden, (soweit notwendig) ein Knoten v_i mit dem Etikett *LINKS*(p) und ein Knoten v_j mit dem Etikett z erzeugt und dann v_j mit v_i und v_i mit allen Knoten aus Z_i verbunden.

PROZEDUR X-REDUCE

(* Durchführen aller zulässigen Reduktionen *)

METHODE:

Entferne ein Element <v, x, p> aus R.

N $\Leftarrow$ *Links*(p).

Für alle k $\in$ {k | es gibt einen Pfad der Länge 2 * |*Rechts*(p)|-2 von x nach k}:

Für alle z $\in$ {z | $\exists$(y)(y $\in$ *Nachfolger*(k) $\wedge$ *Goto*(*Zustand*(y), N) = z)}:

Z $\Leftarrow$ {z' | z' $\in$ *Nachfolger*(k) $\wedge$ *Goto*(*Zustand*(z'), N) = z}.

Wenn <$\exists$(u)(u $\in$ U_{ij} $\wedge$ *Zustand*(u) = z)>

Dann

Wenn <es gibt keinen Knoten N(w) $\in$ *Nachfolger*(u) $\wedge$ *Nachfolger*(w) = Z>

Dann

Wenn <N(w) $\notin$ *Nachfolger*(u)>

Dann <Erzeuge einen Knoten N(w) in Γ>

<Erzeuge eine Kante von u nach w in Γ>

Für alle z' $\in$ Z:

Wenn <Es gibt keine Kante von w nach z' in Γ>

Dann <Erzeuge eine Kante von w nach z'>

Wenn <u $\notin$ A>

Dann

Für alle q, mit *reduce* q $\in$ *Action*(*Zustand*(u), w_{i+1}) $\wedge$ |*Rechts*(q)| > 0:

<R $\Leftarrow$ R $\cup$ {<u, w, q>}>

Sonst

<Erzeuge zwei Knoten s(u) und N(w) in Γ.>

<Erzeuge eine Kante von u nach w in Γ.>

Für alle z' $\in$ Z:

<Erzeuge eine Kante von w nach z' in Γ.>

A $\Leftarrow$ A $\cup$ {u}.

U_{ij} $\Leftarrow$ U_{ij} $\cup$ {u}.

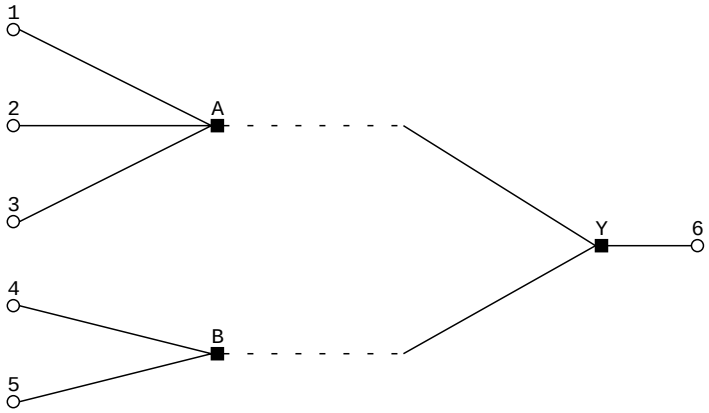
Beispiel (11-7)
Wenn Γ die folgende Form hat und es gilt:

(a) R = {<6, y, p>}

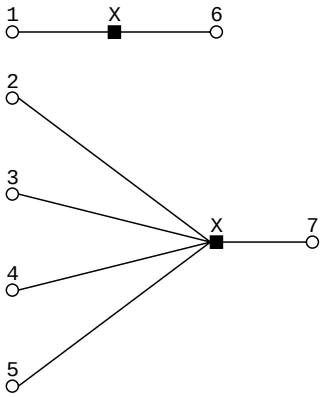
(b) p = X $\rightarrow \alpha$

(c) GOTO(1, X) = 6

(d) GOTO(2, X) = GOTO(3, X) = GOTO(4, X) = GOTO(5, X) = 7



und die Länge des Pfades von A bzw. B nach Y sei 2 * | α |, dann erhalten wir:



E-REDUCE. Durch die Ausführung einer Reduce-Operation mit einer Tilgungsregel werden keine Knoten aus Γ entfernt, sondern ähnlich wie bei Shift-Operationen neue Knoten generiert: Für jeden Zustand, in den der Parser durch Ausführung einer Reduce-Operation mit einer Tilgungsregel p übergehen kann, wird ein Knoten in Γ gebildet, von dem aus eine Kante zu einem neuen, die linke Seite von p repräsentierenden Knoten führt. Dieser Knoten wird mit allen Knoten verbunden, die Zustände des Parsers vor dieser Reduce-Operation repräsentieren.

PROZEDUR E-REDUCE
(* Durchführen aller zulässigen Reduktionen mit e-Regeln *)

METHODE:
$U_{i,j+1} \Leftarrow \emptyset$.
Für alle Zustände z, für die es ein <v, p> $\in R_e$ gibt,
mit Goto(*Zustand*(v), *Links*(p)) = z:
<N $\Leftarrow$ *Links*(p)>
<Erzeuge zwei Knoten z(k) und N(k') in Γ>
<Erzeuge eine Kante von k nach k' in Γ>
$<U_{i,j+1} \Leftarrow U_{i,j+1} \cup \{k\}>$
Für alle <v, p> $\in R_e$ mit Goto(*Zustand*(v), *Links*(p)) = z:
<Erzeuge eine Kante von k' nach v in Γ>
$R_e \Leftarrow \emptyset$.
$A \Leftarrow U_{i,j+1}$.
$j \Leftarrow j+1$.

11.2.2 Der Parsingalgorithmus

Wir verwenden die in 11.1.4 beschriebenen Verfahren des *local ambiguity packing* und *subtree sharing*, um eine möglichst effiziente Repräsentation für die Strukturbeschreibungen eines Satzes zu erhalten: Für jeden Satz wird ein *Parse-Wald* („shared packed forest“) generiert, der alle Strukturbeschreibungen des Satzes in komprimierter Form repräsentiert. Ein *Parse-Wald* kann als ein generalisierter geordneter gerichteter Graph aufgefaßt werden.

Definition (11-1) *gerichteter Graph*
Ein gerichteter Graph $D_g = <V, E>$ besteht aus einer Menge von Knoten V und aus einer Menge von Tupeln E. Jedes Element aus E hat die Form:
$$<v, <w_1, \ldots, w_n>>, \text{ mit } v, w_i \in V.$$
Die *Nachfolgerliste* $<w_1, \ldots, w_n>$ enthält die Knoten des Graphen, zu denen von v eine Kante führt. Die erste Kante führt nach w_1, die zweite nach w_2, ... etc.

In einem (einfachen) geordneten gerichteten Graphen gibt es für jeden Knoten höchstens eine Nachfolgerliste; in einem generalisierten geordneten gerichteten Graphen dagegen kann es für einen Knoten mehr als eine Nachfolgerliste geben.

Neben einer globalen Variablen T zur Verwaltung des *Parse-Waldes* benötigen wir zwei Funktionen auf generalisierten geordneten gerichteten Graphen:

- *NACHFOLGER_LISTEN(v)* nimmt einen Knoten v von T als Argument und liefert die Liste aller Nachfolgerlisten von v in T als Wert.
- *NEUE_NACHFOLGER(v, L)* nimmt einen Knoten v von T und eine Liste von Nachfolgern L als Argumente und ergänzt T mit <v, L>.

Beispiel (11-8)
Wenn P = {<a, <b, c>>, <a, <d, e>>, <b, <f, g>>}, dann gilt:

(a) NACHFOLGER(a) = {<a, <b, c>>, <a, <d, e>>} und

(b) NEUE_NACHFOLGER(a, <f, h>) führt dazu, daß P = {<a, <b, c>>, <a, <d, e>>, <a, <f, h>>, <b, <f, g>>}.

Die Erweiterung des Erkennungsalgorithmus zu einem Parsingalgorithmus legt es nahe, als Etiketten der Symbolknoten des Stacks nicht mehr Kategoriensymbole der Syntax zu verwenden, sondern die Knoten des *Parse-Waldes*, die Wurzeln der mit ihnen korrespondierenden Strukturbeschreibungen sind. Trotz der neuen Etikettierung der Symbolknoten sind (außer bei der Prozedur X-REDUCE) nur geringe Änderungen der im letzten Abschnitt definierten Prozeduren erforderlich:

1. In TOMITA_PARSE ist neben dem Stack Γ auch der *Parse-Wald* T zu initialisieren ($T \Leftarrow \emptyset$) und nach erfolgreicher Analyse eines Satzes statt eines booleschen Wertes der *Parse-Wald* zurückzugeben.

2. VERARBEITE-WORT kann unverändert übernommen werden.

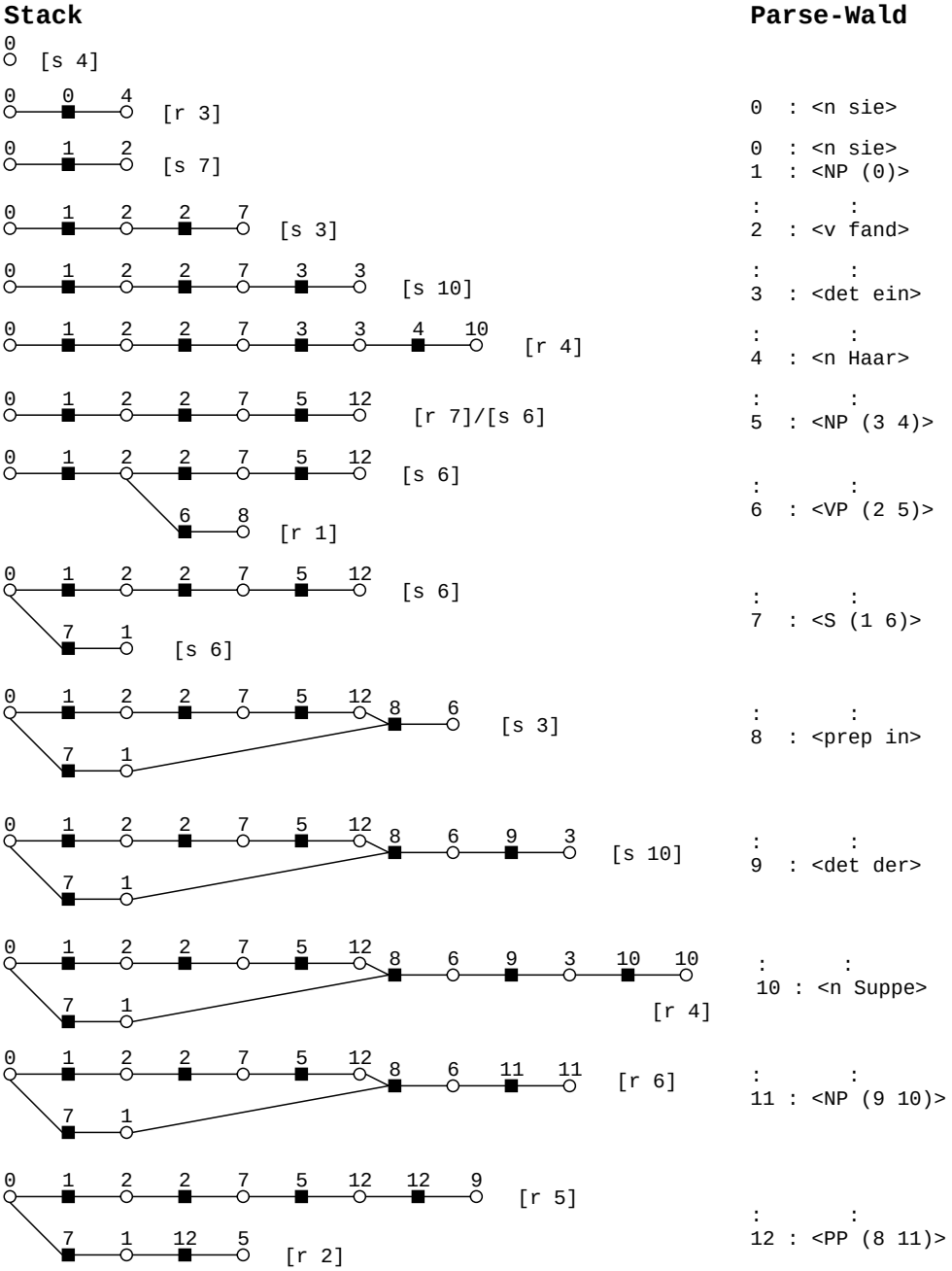
3. Wenn ein Feld der Parsingtabelle das ausgezeichnete Symbol *accept* enthält, ist RESULTAT die Wurzel des *Parse-Waldes* zuzuweisen (VERARBEITE_TABELLE):
   Wenn $<\alpha = accept>$
   Dann $<RESULTAT \Leftarrow v>$.

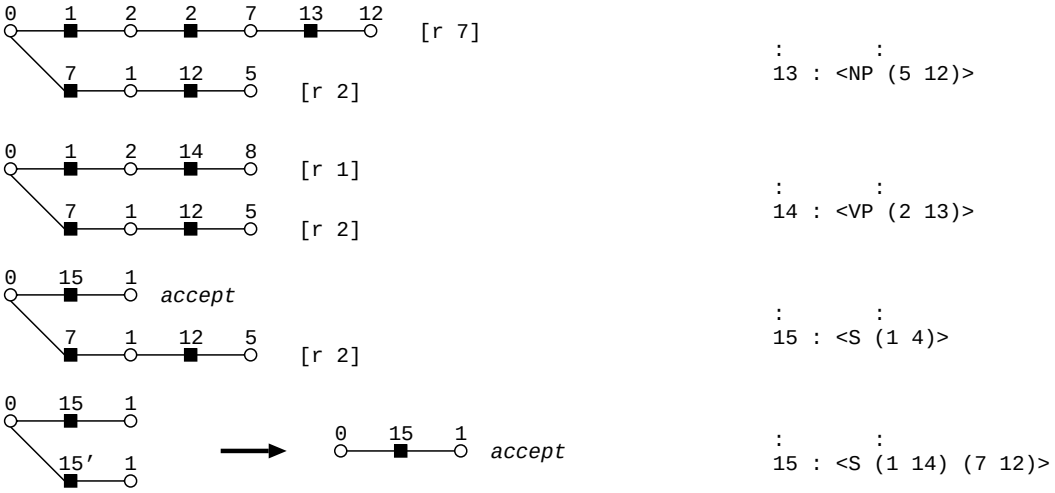
4. Bei Ausführung von Shift-Operationen (X-SHIFT) wird in T ein mit dem nächsten Wort des Satzes etikettierter Knoten erzeugt, der als Etikett für alle zu generierenden Symbolknoten verwendet wird:
   Erzeuge einen Knoten n mit dem Etikett w_{i+1} *in T*
   Für alle z, mit <v, z> ∈ Q:
   Erzeuge zwei Knoten z(k) und n(k') in Γ.

5. In E-REDUCE sind nach „N ⇐ *LINKS*(p)“ die folgenden Zeilen einzufügen:
   Erzeuge in T einen Knoten n mit dem Etikett N
   NEUE_NACHFOLGER(n,<>)
   Erzeuge zwei Knoten z(k) und n(k') in Γ.

Beispiel (11-9)

Ausgehend von der Syntax aus Beispiel (11-1) und einem geeigneten Lexikon erhalten wir für den Satz „Sie fand ein Haar in der Suppe“ folgenden Trace:





Da die Änderung in X-REDUCE etwas umfassenderer Art sind, geben wir die geänderte Prozedur vollständig an:

PROZEDUR X-REDUCE*

METHODE:

Entferne ein Element <v, x, p> aus R.

N $\Leftarrow$ *Links*(p).

Für alle k $\in$ {k | es gibt einen Pfad der Länge 2 * |*Rechts*(p)|-2 von x nach k}:

L $\Leftarrow$ <*Symbol*(r_1),..., *Symbol*($r_{|Links(p)|}$)>, mit z_1 = x, $z_{|Links(p)|}$ = y und z_2, ..., $z_{|Links(p)|-1}$ sind die Symbolknoten des Pfades von x nach y.

Für alle z $\in$ {z| $\exists$(y)(y $\in$ *Nachfolger*(k) $\wedge$ *Goto*(*Zustand*(y), N) = z)}:

Z $\Leftarrow$ {z' | z' $\in$ *Nachfolger*(k) $\wedge$ *Goto*(*Zustand*(z'), N) = z}.

Wenn <$\exists$(u)(u $\in$ U_{ij} $\wedge$ *Zustand*(u) = z)>

Dann

Wenn <$\exists$(N(w))(N(w) $\in$ *Nachfolger*(u) $\wedge$ *Nachfolger*(w) = Z>

Dann <*Neue_Nachfolger(Symbol(w), L)*>

Sonst <Erzeuge einen Knoten n mit dem Etikett N in T.>

<*Neue_Nachfolger(n, L).*>

Wenn <N(w) $\notin$ *Nachfolger*(u)>

Dann <Erzeuge einen Knoten n(w) in Γ>

<Erzeuge eine Kante von u nach w in Γ>

Für alle z' $\in$ Z:

Wenn <Es gibt keine Kante von w nach z' in Γ>

Dann <Erzeuge eine Kante von w nach z'>

```
                Wenn   <u ∉ A>
                Dann
                       Für alle q, mit reduce q ∈ Action(Zustand(u), w_{i+1})
                       ∧ |Rechts(q)| > 0: <R ⇐ R ∪ {<u, w, q>}>
Sonst           <Erzeuge einen Knoten n mit dem Etikett N in T.>
                <Neue_Nachfolger(n, L).>
                <Erzeuge zwei Knoten s(u) und n(w) in Γ.>
                <Erzeuge eine Kante von u nach w in Γ.>
                Für alle z' ∈ Z:
                   <Erzeuge eine Kante von w nach z' in Γ.>
                A ⇐ A ∪ {u}.
                U_ij ⇐ U_ij ∪ {u}.
```

11.3 Komplexität

Tomitas Parsingalgorithmus basiert auf Knuths Parsingalgorithmus für LR(k)-Syntaxen. Knuths Algorithmus ist äußerst effizient: Er erlaubt eine Verarbeitung von LR(k)-Syntaxen in linearer Zeit ($O(n^3|G|)$, mit $|G|$ als der Zahl der Symbole von G). M. Johnson hat gezeigt [**?**], daß der Tomita-Algorithmus in seiner ursprünglichen Form nicht, wie von Tomita angenommen, die gleiche Zeitkomplexität besitzt wie der Earley-Algorithmus ($O(n^3|G|^2)$) bzw. in vielen Fällen sich effizienter verhält als der Earley-Algorithmus, sondern es Grammatiken gibt, die er exponentiell langsamer verarbeitet. J. R. Kipps hat eine Modifikation des Tomita-Algorithmus vorgeschlagen [**?**], die sicherstellt, daß die Zeitkomplexität des Algorithmus für beliebige Grammatiken $O(n^3|G|^2)$ beträgt.

11.4 Implementierung

Wir beschränken uns auf eine Implementierung des Erkennungsalgorithmus in Lisp.

Deklarationen

Wir setzen voraus, daß auf alle zum Parsen erforderlichen Informationsquellen (Syntax, Lexikon und Parsingtabelle) über globale Variablen zugegriffen werden kann:

```
(defvar *syntax*)        (defvar *action*)
(defvar *lexikon*)       (defvar *goto*)
```

Die beiden anderen von uns verwendeten globalen Variablen dienen zur Verwaltung des Stacks und eines Zeigers auf den zuletzt generierten Knoten:

```
(defvar *stack*)         (defvar *stack-zaehler*)
```

Zur Repräsentation des graph-strukturierten Stacks verwenden wir einen Vektor, in dessen Feldern sukzessive die neu generierten Knoten des Stacks eingetragen werden. Ein Eintrag K = <Etikett, Nachfolger> besteht aus dem Etikett des generierten Knotens (ein Zustands- oder Kategoriensymbol) und der Menge der Indizes der Tochterknoten des neu generierten Knotens. Zunächst besteht der Graph nur aus dem Knoten, der den Startzustand des Parsers repräsentiert: Das Feld mit dem Index 0 des Vektors enthält den Eintrag <0, ∅>. Da nur neue Einträge hinzugefügt und keine alten Einträge gelöscht werden, läßt sich aus den im Vektor gespeicherten Informationen die Verarbeitungsgeschichte des Parsers rekonstruieren.

Selektoren & Prädikate

```
; für die Parsingtabelle[8]:

;;; STARTZUSTAND
;;; Liefert den Startzustand des Parsers.
(defmacro startzustand ()
  `(caar *action*))

; für den Stack:

;;; ETIKETT
;;; Liefert das Etikett eines Knotens.
(defmacro etikett (knoten)
  `(first (aref *stack* ,knoten)))

;;; NACHFOLGER
;;; Liefert die Liste aller Knoten, zu denen vom Knoten KNOTEN eine Kante führt.
(defmacro nachfolger (knoten)
  `(second (aref *stack* ,knoten)))

;;; GENERIERE-KNOTEN
;;; Erzeugt einen neuen Zustands-/Symbolknoten mit dem Etikett k und (optional)
;;; einen Verweis auf alle Tochterknoten.
(defmacro generiere-knoten (k &optional nachfolger)
  `(setf *stack-zaehler* (1+ *stack-zaehler*)
         (aref *stack* *stack-zaehler*)
         (list ,k (or ,nachfolger (list (1- *stack-zaehler*))))))

; für die Syntax:

;;; E-REGEL-P
;;; Das Prädikat testet, ob REGEL eine Tilgungsregel ist.
(defmacro e-regel-p (regel)
  `(endp (rest ,regel)))
```

[8] ACTION und GOTO: vgl. Kaptitel 10

```
;;; ITE-REGEL
;;; Liefert die Regel mit dem Index I.
(defmacro ite-regel (i)
  `(nth (1- ,i) (regeln)))
```

Hauptfunktionen

```
;;; TOMITA-RECOGNIZE
;;; Hauptfunktion
;;; Die Variablen EINGABE, RESULTAT, U, A, R, Re und Q sind special-Variablen.
;;; EINGABE   - Liste von Listen lexikalischer Kategorien;
;;; RESULTAT  - boolesche Variable;
;;; U         - Menge der nach der Verarbeitung des letzten Wortes erzeugten
;;;             Knoten;
;;; A         - Menge der aktiven Knoten und
;;; R, Re, Q  - Menge von Einträgen, die die Reduktionsoperationen ohne/mit Tilgung-
;;;             sregeln und Shift-Operationen steuern.
(defun tomita-recognize (satz)
  (progv '(eingabe resultat u a r re q)
         `(,(kategorien-zuweisen satz) nil (0) nil nil nil nil)
    (initialisiere-stack)
    (dotimes (i (length eingabe) resultat)
      (verarbeite-wort))))

;;; VERARBEITE-WORT
(defun verarbeite-wort ()
  (setq a u r nil re nil q nil)
  (loop
    (when (and (endp a) (endp r) (endp re)) (return (shift)))
    (cond (a (tabelle-auswerten))
          (r (reduce))
          (re (e-reduce)))))

;;; TABELLE-AUSWERTEN
;;; Für jeden aktiven Knoten und jede dem nächsten Wort zugeordnete lexikalische
;;; Kategorie werden alle Einträge des korrespondierenden Feldes der Action-Tabelle
;;; ausgewertet.
(defun tabelle-auswerten ()
  (loop
    (when (endp a) (return))
    (auswerten (pop a) (first eingabe))))
```

```
;;; AUSWERTEN
(defun auswerten (knoten kategorien)
  (dolist (kategorie kategorien)
    (dolist (aktion (action (etikett knoten) kategorie))
      (cond ((eq aktion 'accept) (setq resultat t))
            ((eq (first aktion) 's)
             (setq q (neue-shift-zustaende (second aktion) knoten kategorie)))
            ((and (eq (first aktion) 'r) (not (e-regel-p (ite-regel (second aktion)))))
             (dolist (x (nachfolger knoten))
               (push `(,knoten ,x ,(second aktion)) r)))
            (t (push `(,knoten ,(second aktion)) re))))))

;;; SHIFT
;;; Jeder Eintrag in Q hat das Format:
;;; <neuer-Zustand (alter-Zustand-Index*) lexikalische Kategorie>.
(defun shift ()
  (setq u nil)
  (dolist (eintrag q)
    (generiere-knoten (third eintrag) (second eintrag))
    (generiere-knoten (first eintrag))
    (push *stack-zaehler* u))
  (setq q nil eingabe (rest eingabe)))

;;; NEUE-SHIFT-ZUSTAENDE
;;; Erzeugt neue Einträge in Q.
(defun neue-shift-zustaende (neu alt kat)
  (let ((eintrag (assoc neu q)))
    (if (and eintrag (eq (third eintrag) kat))
        (cons (list neu (adjoin alt (second eintrag)) kat)
              (remove eintrag q :test #'equal))
        (cons (list neu (list alt) kat) q))))

;;; E-REDUCE
;;; Format der Einträge in Re: <neuer-Zustand (alter-Zustand-Index*) Regelindex>.
(defun e-reduce ()
  (setq u nil)
  (dolist (eintrag re)
    (generiere-knoten (third eintrag) (second eintrag))
    (generiere-knoten (first eintrag))
    (push *stack-zaehler* u))
  (setq re nil a u))
```

```
;;; NEUE-E-REDUCE-KNOTEN
;;; Erzeugt in Re neue Einträge.
(defun neue-e-reduce-knoten (alt regelindex)
  (let* ((n (first (ite-regel regelindex)))
         (neu (goto (etikett alt) n))
         (eintrag (assoc neu re)))
    (if (and eintrag (eq (third eintrag) n))
        (cons (list neu (adjoin alt (second eintrag)) n)
              (remove eintrag re :test #'equal))
        (cons (list neu (list alt) n) re))))

;;; REDUCE
;;; Jeder Eintrag in R hat das Format:
;;; <alter-Zustand-Index Symbolknoten-Vorgänger-Index Regel-Index>
;;; Es werden folgende lokale Variablen verwendet:
;;; N      - linke Seite der zur Reduktion verwendeten Regel;
;;; K      - letzter zu entfernender Symbolknoten;
;;; PAAR - (Zustand-nach-Reduktion (Zustände-vor-Reduktion)) und
;;; V      - ein Knoten aus U.
(defun reduce ()
  (let ((n nil))
    (dolist (eintrag r)
      (setq n (first (ite-regel (third eintrag))))
      (dolist (k (reduce-knoten (list (second eintrag))
                                (reduce-laenge (third eintrag))))
        (dolist (paar (neue-reduce-zustaende (nachfolger k) n))
          (let ((v (finde-knoten (first paar) u)))
            (cond (v (reduce1 v n paar))
                  (t (generiere-knoten n (second paar))
                     (generiere-knoten (first paar))
                     (setq a (adjoin *stack-zaehler* a)
                           u (adjoin *stack-zaehler* u)))))))))
  (setq r nil))

;;; REDUCE1
;;; Verarbeitet den Fall, daß es bereits einen Knoten gibt, dessen Etikett mit dem zu
;;; generierenden Knoten identisch ist.
(defun reduce1 (v n paar)
  (let ((w (finde-knoten n (nachfolger v))))
    (unless (and w (equal (nachfolger w) (second paar)))
      (cond (w (setf (second (aref *stack* w))
                     (union (nachfolger w) (second paar))))
```

```
            (t (generiere-knoten n (second paar))
               (setf (second (aref *stack* v))
                     (adjoin *stack-zaehler* (nachfolger v)))))
      (unless (member v a)
        (dolist (kategorie (first eingabe))
          (dolist (eintrag (action (first paar) kategorie))
            (when (and (eq (first eintrag) 'r)
                       (not (e-regel-p (ite-regel (second eintrag)))))
              (push (list v n (second eintrag)) r))))))))
```

Hilfsfunktionen

```
;;; INITIALISIERE-STACK
;;; Generiert den Stack und trägt einen mit dem Startzustand des Parsers korrespon-
;;; dierenden Knoten ein.
(defun initialisiere-stack ()
  (setf *stack* (make-array 100 :initial-element nil)
        (aref *stack* 0) (list (startzustand) nil) *stack-zaehler* 0))

;;; KATEGORIEN-ZUWEISEN
;;; Liefert eine Liste, die für jedes Wort aus Satz die Liste aller ihm durch das Lexikon
;;; zugewiesenen Kategorien enthält.
(defun kategorien-zuweisen (satz)
  (let ((resultat nil) (kategorien nil))
    (dolist (wort satz (append resultat '(($))))
      (if (setq kategorien (eintrag-kategorien wort))
          (setf resultat (append resultat (list kategorien)))
          (return (format t "~%Fuer das Wort: ~a existiert kein Lexikoneintrag!~%"
                          wort))))))

;;; FINDE-KNOTEN
;;; Sucht in K-MENGE nach einem Knoten mit dem Etikett NAME.
(defun finde-knoten (name k-menge)
  (cond ((endp k-menge) nil)
        ((eq name (etikett (first k-menge))) (first k-menge))
        (t (finde-knoten name (rest k-menge)))))

;;; REDUCE-KNOTEN
;;; Liefert die Menge aller Symbolknoten, die bei einer Reduce-Operation im letzten
;;; Schritt noch entfernt werden müssen.
(defun reduce-knoten (knoten laenge)
  (if (= laenge 0) knoten
      (reduce-knoten (berechne-neue-knoten knoten) (- laenge 2))))
```

```
;;; BERECHNE-NEUE-KNOTEN
;;; Berechnet alle von den Symbolknoten SKNOTEN aus erreichbaren Symbolknoten.
;;; (Nachfolger der von ihnen dominierten Zustandsknoten)
(defun berechne-neue-knoten (sknoten)
  (let ((z-nachfolger nil) (s-nachfolger nil))
    (dolist (s sknoten)
      (setq z-nachfolger (union z-nachfolger (nachfolger s))))
    (dolist (z z-nachfolger s-nachfolger)
      (setq s-nachfolger (union s-nachfolger (nachfolger z))))))

;;; REDUCE-LAENGE
;;; Berechnet die Zahl der aus dem Stack zu entfernenden Knoten.
(defun reduce-laenge (i)
  (- (* (length (rest (ite-regel i))) 2) 2))

;;; NEUE-REDUCE-ZUSTAENDE
;;; Berechnet eine Liste von Paaren:
;;; erstes Element  - Zustand Z,
;;; zweites Element - Liste von Zuständen, die durch Reduktion zu Z führen.
(defun neue-reduce-zustaende (knoten symbol)
  (let ((paare nil) (zustand nil))
    (dolist (k knoten paare)
    (if (assoc (setf zustand (goto (etikett k) symbol)) paare)
        (push k (second (assoc zustand paare)))
        (when zustand (push `(,zustand (,k)) paare))))))
```

Aufgaben

11.1 Warum können mit dem Tomita-Algorithmus keine unbeschränkt ambigen und keine zyklischen Syntaxen verarbeitet werden?

11.2 Erweitern Sie den in Lisp implementierten Tomita-Recognizer zu einem Parser.

11.3 Implementieren Sie den Tomita-Algorithmus in Prolog.

11.4 Entwickeln Sie einen tomita-basierten On-line-Parser. Ein On-line-Parser ist ein Parser, der mit der Analyse eines Satzes beginnt, sobald ein Wort eingelesen wurde, und nicht erst wartet, bis die Eingabe des Satzes abgeschlossen wurde.

11.5 Modifizieren Sie den Tomita-Algorithmus so, daß er auch komprimierte Parsingtabellen, die **any**-Einträge enthalten, verarbeiten kann.

Teil V

Unifikationsbasierte Grammatikformalismen

Alle der in den ersten drei Teilen dieses Buchs beschriebenen Algorithmen wurden ursprünglich für die Verarbeitung einfacher kontextfreier Syntaxen konzipiert. Es stellt sich aus Sicht der (Computer-)Linguistik die Frage, ob Syntaxen dieses Typs sich nicht nur zur Beschreibung formaler Sprachen wie z.B. Programmiersprachen, sondern auch zur Beschreibung natürlicher Sprachen eignen. Unter dem Stichwort der „Kontextfreiheit natürlicher Sprache“ sind in den letzten 40 Jahren zwei Fragen heftig diskutiert worden[9]:

1. Lassen sich alle Konstruktionen, die natürliche Sprachen aufweisen, *prinzipiell* durch kontextfreie Syntaxen beschreiben?

2. Ist es möglich, für alle diese Konstruktionen *linguistisch adäquate Analysen* mit kontextfreien Syntaxen zu formulieren?

Die erste der beiden Fragen kann als immer noch offen betrachtet werden, auch wenn z.Z. die Meinung überwiegt, durch Arbeiten wie die von [**?**], [**?**] und [**?**] sei endgültig belegt worden, daß es zumindest in einigen natürlichen Sprachen Konstruktionen gäbe, die sich grundsätzlich nicht durch kontextfreie Syntaxen beschreiben ließen. Diese Überzeugung motiviert das Interesse an *mild kontextsensitiven* Grammatikformalismen, wobei der Begriff der milden Kontextsensitivität charakterisiert wird durch die folgenden drei Eigenschaften[10]:

1. begrenzte Fähigkeit zur Beschreibung sich überschneidender Abhängigkeitsbeziehungen („cross-serial dependencies“),

2. lineares Wachstum der Satzlänge[11] und

3. Verfügbarkeit polynomialer Parsingalgorithmen.

Die zweite Frage dagegen wird seit geraumer Zeit von der überwiegenden Zahl aller (Computer-)LinguistInnen verneint. Dieser relativ deutliche Konsens basiert auf der Beobachtung, daß kontextfreie Syntaxen nicht in der Lage sind, die in natürlichen Sprachen bestehenden Kookkurrenzrestriktionen adäquat abzubilden. Als in dieser Hinsicht besonders problematische Fälle gelten:

1. Kongruenz und Rektion;

2. Wortstellung und

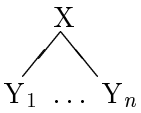
3. nicht-lokale Abhängigkeiten (s.o.).

[9] Die Diskussion litt z.T. erheblich unter dem Versäumnis, diese beiden Fragen nicht klar voneinander zu trennen.

[10] Vgl. [**?**, S.225].

[11] Wenn man die Sätze einer kontextfreien bzw. mild kontextsensitiven Sprache der Länge nach sortiert, dann zeigt sich, daß die Länge der Sätze linear wächst: Sprachen, die diese Eigenschaft nicht aufweisen, wie z.B. $L_1 = \{a^{n^2} | n \geq 0\}$ und $L_2 = \{a^{2^n} | n \geq 0\}$, sind nicht mild kontextsensitiv.

Betrachten wir, um dieses Argument besser zu verstehen, die Restriktionen, die die Verwendung kontextfreier Regeln der Analyse natürlicher Sprache auferlegt: Aus linguistischer Perspektive können kontextfreie Regeln als Kriterien für die Zulässigkeit von Konstituenten betrachtet werden; d.h. eine Regel $X \rightarrow Y_1 \ldots Y_n$ $(n \geq 0)$ sanktioniert Konstituenten der Form:



Welche Eigenschaften sind für kontextfreie Regeln charakteristisch?

1. Jede Regel legt fest, aus welchen Kategorien eine Konstituente besteht $(X, Y_1, \ldots, Y_n)$.
2. Es wird eine Kategorie (X) ausgezeichnet, die die übrigen Kategorien (*direkt*) dominiert.
3. Die Reihenfolge der von X dominierten Kategorien wird fixiert (Y_i vorY_j, mit j > i).
4. Kategorien werden als *atomare*, nicht weiter analysierbare Objekte behandelt.

Diese Eigenschaften sind dafür verantwortlich, daß sich in einfachen kontextfreien Syntaxen weder Generalisierungen bzgl. des Vorkommens von Kategorien innerhalb einer Konstituente (Kongruenz und Rektion) bzw. verschiedener Konstituenten (nicht-lokale Abhängigkeiten) noch Generalisierungen bzgl. der linearen Abfolge von Kategorien (Wortstellung) explizit repräsentieren lassen[12]:

1. Kongruenz und Rektion

   (a) $NP \rightarrow det\ n$ $\Longrightarrow$ der Baum, ∗die Baum, ∗das Baum

   $$\Downarrow$$

   $$\left\{ \begin{array}{l} NP \rightarrow det_{mas}\ n_{mas} \\ NP \rightarrow det_{fem}\ n_{fem} \\ NP \rightarrow det_{neu}\ n_{neu} \end{array} \right\}$$

   (b) $PP \rightarrow p\ NP$ $\Longrightarrow$ auf [dem Baum], ∗auf [der Baum], ∗auf [des Baums], auf [den Baum]

   $$\Downarrow$$

   $$\left\{ \begin{array}{l} PP \rightarrow p_{gen}\ NP_{gen} \\ PP \rightarrow p_{dat}\ NP_{dat} \\ PP \rightarrow p_{akk}\ NP_{akk} \end{array} \right\}$$

[12] Vgl. [?].

2. Wortstellung
   Für das Deutsche ist es charakteristisch, daß die nominalen Kategorien relativ frei positionierbar sind („partial ordering freedom"). So sind z.B. die folgenden Sätze syntaktisch wohlgeformt[13]:

   Der Minister hat dem Syntaktiker das Bundesverdienstkreuz verliehen.
   $(S \rightarrow NP_{nom}\ VP,\ VP \rightarrow V\ NP_{dat}\ NP_{akk})$
   Der Minister hat das Bundesverdienstkreuz dem Syntaktiker verliehen.
   $(S \rightarrow NP_{nom}\ VP,\ VP \rightarrow V\ NP_{akk}\ NP_{dat})$
   Das Bundesverdienstkreuz hat der Minister dem Syntaktiker verliehen.
   $(S \rightarrow NP_{akk}\ VP,\ VP \rightarrow V\ NP_{nom}\ NP_{dat})$
   Das Bundesverdienstkreuz hat dem Syntaktiker der Minister verliehen.
   $(S \rightarrow NP_{akk}\ VP,\ VP \rightarrow V\ NP_{dat}\ NP_{nom})$
   Dem Syntaktiker hat der Minister das Bundesverdienstkreuz verliehen.
   $(S \rightarrow NP_{dat}\ VP,\ VP \rightarrow V\ NP_{nom}\ NP_{akk})$
   ? Dem Syntaktiker hat das Bundesverdienstkreuz der Minister verliehen. $(S \rightarrow NP_{dat}\ VP,\ VP \rightarrow V\ NP_{akk}\ NP_{nom})$

3. Nicht-lokale Abhängigkeiten

   (a) Im morpho-syntaktischen Bereich: trennbare Verbpräfixe
   kontinuierliche Vorkommen
   ..., weil die Post heute keine Paketsendungen zustellt.
   Heute werden von der Post keine Paketsendungen zugestellt.
   diskontinuierliche Vorkommen
   Die Post stellt heute keine Paketsendungen zu.
   (*Verbstamm und Präfix klammern den Verbalkomplex*)
   Stellt die Post heute keine Paketsendungen zu?
   (*Verbstamm und Präfix klammern den Satz*)

   (b) Im syntaktischen Bereich: periphrastische Konstruktionen
   kontinuierliche Vorkommen
   ..., weil die Post Frau M. ein Paket zustellen wird.
   (*Verbale Kategorien nicht getrennt*)
   Die Post wird Frau M. ein Paket zustellen.
   (*Verbale Kategorien durch nominale getrennt*)

[13] Frei nach [?, S.887]. Zur Beschreibung dieser Sätze geeignete kontextfreie Regeln folgen in Klammern.

diskontinuierliche Vorkommen

Frau M. wird die Post ein Paket zustellen.
(*Topikalisierte* NP_{dat})
Wird die Post Frau M. ein Paket zustellen?
(*Invertierung von* V_{aux} *und* NP_{nom})

Auch wenn eine Beschreibung dieser Phänomene mit einfachen kontextfreien Regeln prinzipiell möglich ist, so kann sie aus naheliegenden Gründen nicht als adäquat betrachtet werden:

1. Sie führt zu einer Syntax mit einer sehr großen Zahl von Regeln, die sich beim Parsen negativ auf die Verarbeitungszeit (und den Speicherbedarf) auswirkt.

2. Sie verdeckt die den Phänomenen zugrundeliegenden sprachlichen Regularitäten: So werden u.a. die für das Deutsche geltenden Kongruenzprinzipien (z.B. die Übereinstimmung von Artikel und Nomen hinsichtlich Genus, Numerus und Kasus) nicht *explizit* formuliert, sondern *implizit* durch eine große Zahl gleichartiger Regeln abgebildet.

Eine Möglichkeit, diese deskriptiven Schwächen einfacher kontextfreier Syntaxen zu beseitigen, besteht in der **systematischen Nutzung** solcher syntaktischen Merkmale, wie sie im vorangegangenen Beispiel zur Indizierung der Kategoriensymbole verwendet wurden. Merkmale zur Bezeichnung linguistischer Objekt zu verwenden, ist kein neuer Gedanke, wie ein Blick in die Geschichte der modernen Linguistik zeigt:

Phonologie. Die von Jakobson, Fant und Halle [**?**] entwickelte **Theorie der distinktiven Merkmale** verwendet zur Repräsentation von Phonemen binäre Merkmale. Ein Phonembezeichner wie z.B. /b/ wird als Abkürzung für die folgende Menge von Merkmalsspezifikationen aufgefaßt:

$$\left\{\begin{array}{l} +konsonantisch \\ -durativ \\ +anterior \\ -koronal \\ -nasal \\ -lateral \\ +stimmhaft \\ -verzögert \end{array}\right\}$$

Syntax. In der **Standardtheorie** [**?**] verwendete Chomsky ebenfalls binäre (syntaktische[14]) Merkmale zur Bezeichnung lexikalischer Kategorien und zur Formulierung von *Selektionsregeln*, die unter Verwendung von Merkmalen den kategorialen Rahmen spezifizieren, innerhalb dessen eine Kategorie auftreten kann:

[14] Die meisten der von ihm als „syntaktisch“ klassifizierten Merkmale würden heute eher als *semantische* Merkmale betrachtet werden.

- $N \rightarrow [+N, \pm Common]$
  $[+Common] \rightarrow [\pm Count]$
  $\vdots \qquad \vdots$ [?, S.82]
- $[+V] \rightarrow CS/ \left\{ \begin{array}{l} [+\text{Abstract}]\ \text{Aux\_} \\ [-\text{Abstract}]\ \_ \\ \_\text{Det}\ [+\text{Animate}] \\ \_\text{Det}\ [-\text{Animate}] \end{array} \right\}$ [?, S.95]

Die von Kratzer, Pause & Stechow Mitte der siebziger Jahre formulierte „normale Syntax in komplexer Notation für einen Teil des Deutschen“[15] dagegen nutzt syntaktische Merkmale, um gleichartige kontextfreie Regeln durch komplexe Regeln zusammenzufassen:

> Die Numerus-, Genus- und Kasus-Kongruenz von Artikel und Nomen werden zunächst durch die folgenden Regeln ausgedrückt[16]:
>
> $NP_{num(1)gen(1)kas(1)} \rightarrow det_{num(1)gen(1)kas(1)}\ n_{num(1)gen(1)kas(1)}$
> $NP_{num(1)gen(2)kas(1)} \rightarrow det_{num(1)gen(2)kas(1)}\ n_{num(1)gen(2)kas(1)}$
> $\vdots \qquad\qquad \vdots \qquad\qquad \vdots$
>
> Diese Regeln lassen sich zu einer komplexen Regel zusammenfassen:
>
> $NP_{num\ gen\ kas} \rightarrow det_{num\ gen\ kas}\ n_{num\ gen\ kas} \qquad (1,1,1)(2,2,2)(3,3,3)$
>
> Die Zahlentripel der Form (i, j, k) legen fest, daß das i-te Merkmal der ersten Kategorie, das j-te Merkmal der zweiten und das k-te Merkmal der dritten Kategorie gleich spezifiziert sein müssen.

Diese Syntax kann als als Vorläufer der unifikationsbasierten Formalismen betrachtet werden, die seit Ende der siebziger Jahre die Entwicklung der (Computer-)Linguistik entscheidend beeinflußten.

Unifikationsgrammatiken. Für die Grammatiktheorien und -formalismen, die unter der Bezeichnung „Unifikationsgrammatiken“ zusammengefaßt werden, ist charakteristisch, daß sie Merkmalsstrukturen als zentrales deskriptives Mittel verwenden (z.B. zur Repräsentation von Kategorien, syntaktischen Regeln, funktionalen Strukturen, etc.). Zu den wichtigsten Typen von Unifikationsgrammatiken gehören u.a. PATR-II [?], die **F**unctional **U**nification **G**rammar [?], die **L**exical-**F**unctional **G**rammar [?], die **G**eneralized **P**hrase-**S**tructure **G**rammar [?], die **H**ead-Driven **P**hrase-**S**tructure **G**rammar [?], die **C**ategorial **U**nification **G**rammar [?] und die **U**nification **C**ategorial **G**rammar [?]. Außer den kategorialgrammatischen Formalismen (CUG und UCG) besitzen sie alle einen *kontextfreien Kern* bzw. ein *kontextfreies Rückgrat* („context-free backbone“). Sie verwenden entweder

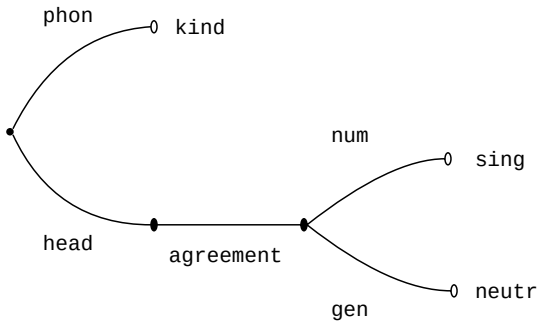
[15] In: [?, Bd.II, S.3-136]

[16] Wir haben die von Kratzer, Pause & Stechow verwendete Notation den von uns in diesem Buch verwendeten Konventionen angepaßt

- annotierte kontextfreie Regeln (LFG) oder
- verschiedene Regeltypen, die sich in kontextfreie Regeln kompilieren lassen (GPSG: ID/LP-Regeln) oder
- spezielle Merkmale bzw. spezielle Merkmale und Regeln, mit denen sich syntaktische Hierarchie- und lineare Abfolgebeziehungen ähnlich wie durch kontextfreie Regeln beschreiben lassen (FUG: *cset/pattern*; HPSG: *subcat*/LP-constraints).

Aus diesem Grund liegt es nahe, zum Parsen von Syntaxen dieses Typs dieselben Algorithmen zu verwenden, die sich für das Parsen einfacher kontextfreier Syntaxen bewährt haben.

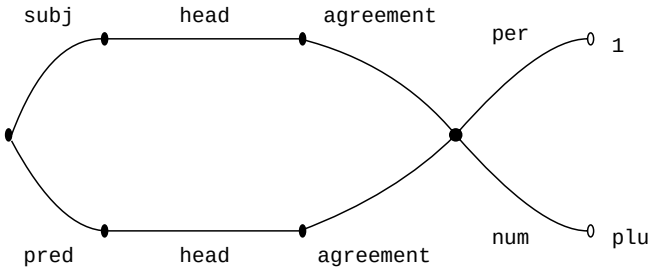
Merkmalsstrukturen. Einfache Merkmalsstrukturen[17] können als gerichtete azyklische Graphen interpretiert werden. Normalerweise werden sie in Form von Attribut-Wert Matrizen (AWMs) notiert:



$$M_1 \left[\begin{array}{ll} phon & kind \\ head & \left[agreement \quad \left[\begin{array}{ll} num & sing \\ gen & neutr \end{array} \right] \right] \end{array} \right]$$

Wenn mehrere Attribute einen *gemeinsamen Wert* haben („re-entrancies"), wird dieser Wert bei Verwendung der AWM-Notation indiziert und nur einmal explizit angegeben. Bei den übrigen Attributen, die diesen Wert aufweisen, wird er durch diesen Index repräsentiert:

[17] Wenn, wie innerhalb der meisten unifikationsbasierten Formalismen gefordert, logische Verknüpfungen wie Disjunktion und Negation in Merkmalsstrukturen zugelassen werden, ist dieser graphentheoretische Ansatz problematisch und eine logikorientierte Rekonstruktion vorzuziehen (vgl. [**?**]).



$$M_2 \begin{bmatrix} subj & \begin{bmatrix} head & \begin{bmatrix} agreement & \boxed{1} \end{bmatrix} \begin{bmatrix} per & 1 \\ num & plu \end{bmatrix} \end{bmatrix} \\ pred & \begin{bmatrix} head & \begin{bmatrix} agreement & \boxed{1} \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

Die Verwendung von *re-entrancies* ermöglicht es z.B., die im Deutschen zwischen Artikel und Nomen bestehenden Kongruenzbeziehungen mit einer Regel zu beschreiben:

$$\begin{bmatrix} cat & NP \\ head & \boxed{1} \begin{bmatrix} agreement & \boxed{2} \end{bmatrix} \end{bmatrix} \longrightarrow \begin{bmatrix} cat & det \\ agreement & \boxed{2} \end{bmatrix} \begin{bmatrix} cat & n \\ head & \boxed{1} \end{bmatrix}$$

Wie diese Beispiele zeigen, gibt es zwei Typen von Attributen: Attribute mit *atomaren* Werten (*phon, num, per, gen, ...*) und Attribute, deren Werte selbst wieder Merkmalsstrukturen sind (*subj, pred, head, agreement, ...*).

Merkmalsstrukturen können als partielle Beschreibungen bestimmter Objekte wie z.B. lexikalischer und syntaktischer Kategorien betrachtet werden. Über der Menge aller Merkmalsstrukturen läßt sich eine reflexive partielle Ordnung durch die Subsumtionsrelation definieren[18]:

Definition *Subsumtion*

Gegeben seien zwei Merkmalsstrukturen M_i, M_j. M_i subsumiert M_j ($M_i \preceq M_j$) gdw.

(i). M_i und M_j atomar sind und $M_i = M_j$ oder

(ii). M_i und M_j komplex sind und es gilt:

(a) Jeder Pfad in M_i ist auch ein Pfad in M_j. Der Wert eines Pfades in M_i subsumiert den Wert des mit ihm in M_j korrespondierenden Pfades.

(b) Alle Pfade, die sich in M_i einen gemeinsamen Wert teilen, teilen sich auch in M_j einen Wert.

Intuitiv betrachtet kann man sagen, daß wenn $A \preceq B$ gilt, die Merkmalsstruktur A weniger Informationen enthält als die Merkmalsstruktur B. Die Merkmalsstruktur

[18] Die zugehörige duale Relation wird als *Extension* bezeichnet.

mit dem geringstmöglichen Informationsgehalt ist die leere Merkmalsstruktur [ ]. Es gilt u.a.:

$$[\,] \preceq \left[\begin{array}{ll} phon & kind \end{array}\right] \preceq \left[\begin{array}{ll} phon & kind \\ head & \left[\begin{array}{ll} agreement & [\,] \end{array}\right] \end{array}\right] \preceq M_1$$

Merkmalsstrukturen, die kompatible Informationen repräsentieren, können miteinander verknüpft werden. Das Resultat ist eine Merkmalsstruktur, die alle Informationen und nur die Informationen enthält, die in den beiden kombinierten Merkmalsstrukturen enthalten sind.

$$M_1 \;\sqcup\; M_3 \left[\begin{array}{ll} cat & NP \\ head & \left[\begin{array}{ll} agreement & \left[\begin{array}{ll} kas & nom \end{array}\right] \end{array}\right] \end{array}\right]$$

$$= \; M_4 \left[\begin{array}{ll} cat & NP \\ phon & kind \\ head & \left[\begin{array}{ll} agreement & \left[\begin{array}{ll} num & sing \\ gen & neutr \\ kas & nom \end{array}\right] \end{array}\right] \end{array}\right]$$

Die Operation, durch die zwei Merkmalsstrukturen miteinander verknüpft werden, wird als *Unifikation* bezeichnet:

Definition *Unifikation*
Die Unifikation zweier Merkmalsstrukturen M_i, M_j ist definiert als die Kategorie M_k ($M_i \sqcup M_j = M_k$), für die gilt:

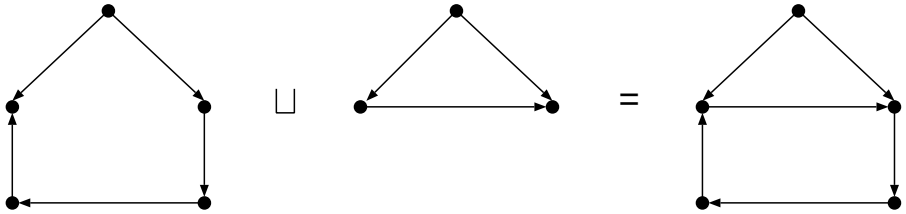
(i). $M_i \preceq M_k$;

(ii). $M_j \preceq M_k$ und

(iii). es gibt keine Merkmalsstruktur M'_k, die die ersten beiden Bedingungen erfüllt und für die gilt: $M'_k \preceq M_k$.

Enthalten die beiden Merkmalsstrukturen inkompatible Informationen, scheitert der Versuch, sie zu unifizieren, bzw. führt zur inkonsistenten Merkmalsstruktur, die zumeist als '$\perp$' notiert wird.

Da die Unifikation zweier azyklischer Graphen zu einem zyklischen Graphen führen kann, müssen Parsingalgorithmen für Unifikationsgrammatiken in der Lage sein, auch zyklische Merkmalsstrukturen korrekt zu verarbeiten. Anderenfalls muß in jedem Schritt die Wohlgeformtheit der durch Unifikation neu gebildeten Merkmalsstrukturen geprüft werden:



$$M_1 \sqcup M_5 \begin{bmatrix} cat & NP \\ head & \begin{bmatrix} agreement & \begin{bmatrix} num & plu \\ kas & nom \end{bmatrix} \end{bmatrix} \end{bmatrix} = \bot$$

da $M_1(head : agreement : num) \neq M_5(head : agreement : num)$

In den folgenden drei Kapiteln beschreiben wir die Grundzüge verschiedener unifikationsbasierter Grammatikformalismen und skizzieren zwei verschiedene Strategien zum Parsen von Unifikationsgrammatiken, die als *direktes* bzw. *indirektes* Parsen von Unifikationsgrammatiken bezeichnet werden können.

Kapitel 12

Definite Clause Grammatiken

Definite Clause Grammatiken (DCGs) sind historisch und systematisch eng mit der Programmiersprache Prolog verbunden, die u.a. zu Zwecken der Verarbeitung natürlicher Sprache entwickelt wurde. DCGs sind im Sprachumfang der meisten Prolog-Implementierungen bereits enthalten, können aber auch leicht mit anderen Prolog-Sprachmitteln vom Benutzer definiert werden. Wegen dieser engen Verzahnung von DCGs und Prolog werden wir darauf verzichten, die Repräsentation und Verarbeitung von DCGs in Lisp zu diskutieren.

12.1 Syntaktische Regeln als logische Ausdrücke

Wir zeigen zunächst, wie sich kontextfreie Regeln durch Hornklauseln repräsentieren lassen, und erläutern anschließend die Beziehung zwischen kontextfreien Regeln und DCG-Regeln, sowie zwischen DCG-Regeln und Hornklauseln.

Eine *Hornklausel* ist eine Disjunktion von Termen, die höchstens einen nicht-negierten Term enthält:

$$\neg y_1 \vee \neg y_2 \ldots \vee \neg y_n \vee x \qquad \equiv \qquad y_1 \wedge y_2 \ldots \wedge y_n \Rightarrow x$$

Hornklauseln bilden die Grundlage der Programmiersprache Prolog. Prolog-Regeln sind nichts anderes als eine bestimmte Repräsentationsform für Hornklauseln:

$$y_1 \wedge y_2 \ldots \wedge y_n \Rightarrow x \qquad \equiv \qquad x \ :- \ y_1, \ y_2, \ \ldots, \ y_n$$

Ein Prolog-Interpreter kann als Theorembeweiser für Hornklausellogik betrachtet werden. Kontextfreie Regeln lassen sich relativ direkt in Hornklauseln übersetzen: Wenn Sätze durch Listen von Prolog-Atomen repräsentiert werden, dann kann jedes nicht-terminale Symbol durch ein einstelliges Prädikat und jedes terminale Symbol durch eine Liste, die dieses Symbol enthält, repräsentiert werden.

Beispiel (12-1)

(a) syntaktische Regeln:

$S \rightarrow NP\ VP$ $\quad\quad$ $s(S)$:-
$\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad$ $np(NP)$, $vp(VP)$,
$\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad$ $append(NP, VP, S)$.

Das *Append*-Ziel in der Prolog-Klausel legt die korrekte lineare Abfolge fest.

(b) lexikalische Regeln:

$pro \rightarrow wir$ $\qquad$ $pro(X)$:- $X = [wir]$. bzw.
$pro([wir])$.

Repräsentiert man kontextfreie Regeln durch Hornklauseln, dann übernimmt ein Theorembeweiser für Hornklausellogik (z.B. der Prolog-Interpreter) die Funktion eines Recognizers bzw. Parsers: Statt nach einer Ableitung für einen Satz $\alpha = w_1 \ \ldots \ w_n \ \ (n \geq 0)$ relativ zu einer gegebenen Syntax G zu suchen ($S^* \Rightarrow \alpha$), wird nach einem Beweis für das Theorem $s([w_1, \ \ldots, \ w_n])$ aus der Menge der Axiome G' gesucht, die aus der Übersetzung der kontextfreien Regeln von G in Hornklauseln resultiert ($G' \vdash s([w_1, \ \ldots, \ w_n])$). Übernimmt man die Beweisstrategie des Prolog-Interpreters unverändert, so ist das Resultat ein von links nach rechts arbeitender Top-down-Recognizer/Parser mit Backtracking. Um andere Kontrollstrukturen zu realisieren, ist entweder ein anderer Compiler für DCG-Regeln zu verwenden (vgl. Matsumotos BUP-System, Kapitel 5) oder die Prolog-Beweisstrategie (z.B. via Meta-Interpreter) zu modifizieren.

Welche Beziehung besteht zwischen DCGs bzw. zwischen kontextfreien Syntaxen und DCGs und Hornklauseln? Vom semantischen Gesichtspunkt aus betrachtet sind die Regeln einer DCG nichts anderes als eine spezielle Notation für eine bestimmten Klasse von Hornklauseln. Sie orientiert sich stark an der für kontextfreie Regeln verwendeten Notation, von der sie sich nur in folgenden beiden Punkten unterscheidet:

- Anstelle einfacher Symbole können in DCG-Regeln komplexe Terme als nichtterminale Symbole verwendet werden, z.B.:
$$s(X), \ s(s(NP, VP)), \ \ldots$$
- Zu jeder Regel können zusätzliche Bedingungen angegeben werden, die erfüllt sein müssen, bevor die Regel angewendet werden kann:
$$np(X) \rightarrow det(Y), \ n(X), \ \{common\_noun(X)\}$$

12.2 Differenzlisten

DCG-Regeln werden von den meisten Prolog-Interpretern automatisch in äquivalente Hornklauseln übersetzt. Aus Gründen der Effizienz werden die Teillisten nicht durch *Append*-Ziele explizit miteinander verknüpft, sondern es werden statt dessen *Differenzlisten* verwendet.

Üblicherweise werden in kontextfreien Phrasenstrukturgrammatiken die folgenden beiden Regeltypen unterschieden:

- syntaktische Regeln der Form $c_0 \rightarrow c_1 \ c_2 \ \ldots \ c_n$ und

- lexikalische Regeln der Form $c_0 \rightarrow w$, mit $w \in V_T$ und $c_i \in V_N$.

Diese Regelschemata lassen sich mit Differenzlistenvariablen wie folgt darstellen:

(a) $c_0(X_0,\ X_n) :-$
$\quad c_1(X_0,\ X_1),$
$\quad c_2(X_1,\ X_2),$
$\quad \ldots$
$\quad c_n(X_{n-1},\ X_n).$

(b) $c_0([w|X],\ X).$

Die Kategorien werden als Prädikate für Ketten aufgefaßt, die hier durch Paare von Differenzlistenvariablen ($X_0,\ X_1,\ \ldots,\ X_n$) repräsentiert werden. Dabei bezeichnet X_0 die Position unmittelbar vor dem ersten Wort und X_n die Position unmittelbar hinter dem letzten Wort einer Kette. Diese durch Anfangs- und Endpunkt angegebene Kette fällt unter ein Prädikat c_0, wenn sie sich vollständig in Teilketten zerlegen läßt, die einen Anfangspunkt X_i und einen Endpunkt X_{i+1} haben und der Kategorie c_{i+1} angehören:

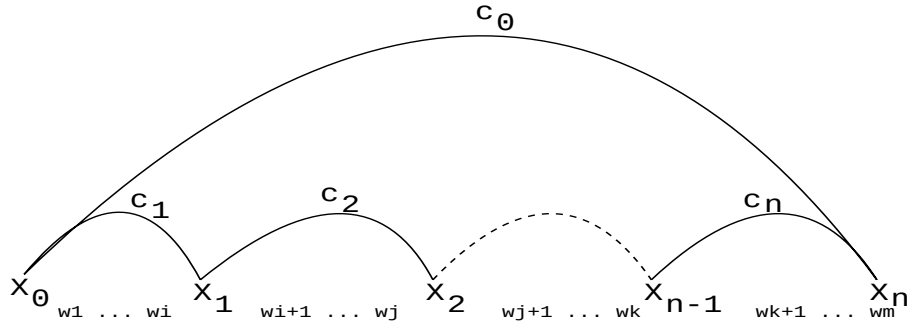


Abbildung (12-1)

So bezeichnet z.B. das Differenzlistenpaar X_1, X_2 in der obigen Abbildung die Teilkette $[w_{i+1} \ldots w_j]$ und das Paar X_0, X_n die Liste $[w_1 \ldots w_m]$. Generell gilt, daß jedem n-stelligen Term einer DCG-Regel ein n+2-stelliges Prolog-Prädikat entspricht:

Beispiel (12-2)

(a) s —> $np,\ vp.$
$s(S0, S2)\ :-\ np(S0, S1),\ vp(S1, S2).$

(b) $s(s(NP, VP))$ —> $np(NP), vp(VP).$
$s(s(NP, VP), S0, S2)\ :-\ np(NP, S0, S1),\ vp(VP, S1, S2).$

(c) pro —> $[wir].$
$pro([wir|S_0], S_0).$

12.3 Vom Recognizer zum Parser

Eine interessante Asymmetrie zwischen kontextfreien Syntaxen und DCGs besteht darin, daß der Übergang vom Satzerkennen zur Satzanalyse bei kontextfreien Syntaxen mit der Verwendung unterschiedlicher Prozeduren (*gleiche Daten / verschiedene Prozeduren*) verbunden ist, während bei DCGs dieselbe Beweisstrategie für das Erkennen bzw. Parsen von Sätzen verwendet werden kann, andererseits aber die Grammatik zu modifizieren ist (*unterschiedliche Daten / gleiche Prozedur*).

Der Grund für diese Asymmetrie liegt darin, daß im ersten Fall die für den Aufbau von Strukturbeschreibungen notwendigen „bookkeeping"-Operationen explizit innerhalb des Parsingalgorithmus spezifiziert werden müssen, bei DCGs dagegen die Regel nur durch zusätzliche Variablen zu erweitern sind und der Aufbau der Strukturbeschreibung via Unifikation keiner weiteren prozeduralen Spezifikation bedarf. In der folgenden Beispiel-DCG werden Strukturbeschreibungen in Form von Bäumen erzeugt, die als Klammerausdrücke repräsentiert werden:

Beispiel (12-3)
Gegeben sei die folgende einfache DCG:

```
s(S0, S2, [s, [NP, VP]]) :-                % S → NP VP
    np(S0, S1, NP),
    vp(S1, S2, VP).
np([hans|R], R, [np, hans]).               % NP → hans
vp([schlaeft|R], R, [vp, schlaeft]).       % VP → schlaeft
```

Die folgende Ableitung zeigt, wie diese DCG durch den Prolog-Interpreter top-down und depth-first verarbeitet wird und die Variablen instantiiert werden:

```
CALL:  s([hans, schlaeft], [ ], X).
CALL:  np([hans, schlaeft], _1, _2).
EXIT:  np([hans, schlaeft], [schlaeft], [np, hans]).
CALL:  vp([schlaeft], [ ], _3).
EXIT:  vp([schlaeft], [ ], [vp, schlaeft]).
EXIT:  s([hans, schlaeft], [ ], [s, [[np, hans], [vp, schlaeft]]]).
```

12.4 Nicht-kontextfreie Erweiterungen

DCGs bilden eine adäquate und naheliegende Möglichkeit, um einen Top-down-Parser mit Depth-first-Suchstrategie für kontextfreie Syntaxen in Prolog zu realisieren. Darüber hinaus erlauben DCGs jedoch auch Erweiterungen, die über die Möglichkeiten von einfachen kontextfreien Syntaxen hinausgehen. Zu den für uns interessantesten Erweiterungsmöglichkeiten zählen:

- Der Aufbau von Strukturbeschreibungsvarianten: Die Strukturbeschreibungen sind prinzipiell unabhängig von der Konstituentenstruktur, wie sie durch das kontextfreie Skelett einer DCG definiert wird.
- Die Einführung von zusätzlichen constraints, die z.B. die Behandlung von Kongruenz, Rektion und semantische Restriktionen erlauben.

Beide Typen von Erweiterungen sollen kurz anhand von einfachen Beispielen skizziert werden.

(a) Zunächst zeigen wir, wie es möglich ist, die Strukturbeschreibungen der DCG aus dem letzten Beispiel ohne Veränderung der Erfüllbarkeitsbedingungen so zu verändern, daß sich differenziertere Strukturbäume ergeben (Beispiel (12-4)), und

(b) in Beispiel (12-5) wird ein zusätzlicher Test auf Numeruskongruenz eingeführt, der mit einer reinen kontextfreien PSG nicht in dieser Form ausgedrückt werden kann.

Beispiel (12-4)

```
s(S0, S2, [s, [NP, VP]]) :-                 % S → NP VP
    np(S0, S1, NP),
    vp(S1, S2, VP).
np([hans|R], R, [np, [n, hans]]).           % NP → n, n → hans
vp([schlaeft|R], R, [vp, [v, schlaeft]]).   % VP → v, v → schlaeft
```

Beispiel (12-5)

```
s(S0, S2, [s, [NP, VP]]) :-
    np(S0, S1, NP, NUM),
    vp(S1, S2, VP, NUM).
np([hans|R], R, [np, hans], sing).
vp([schlaeft|R], R, [vp, schlaeft], sing).
np([hunde|R], R, [np, hunde], plu).
vp([bellen|R], R, [vp, bellen], plu).
```

Durch die beiden Vorkommen derselben Variable NUM in der ersten Klausel wird sichergestellt, daß die NP und die VP eines Satzes hinsichtlich des Merkmals *Numerus* kongruent sind. Dadurch werden ungrammatische Ketten wie „hans bellen“ usw. ausgeschlossen.

Wenn auf diese Weise weitere Merkmale eingeführt werden sollen, dann muß die Stelligkeit der relevanten Prädikate in der im vorangegangenen Beispiel illustrierten Weise angepaßt werden. Da Prolog-Terme mit unterschiedlicher Stelligkeit nicht unifizieren (Termunifikation[1]), ist es notwendig, diese Änderungen in allen Vorkommen der betroffenen Prädikate vorzunehmen; d.h. selbst in den Regeln, in denen diese Merkmale streng genommen irrelevant sind, müssen sie spezifiziert werden. Die Verwendung von Termunifikation hat in diesem Zusammenhang verschiedene Nachteile:

1. DCG-Regeln in großen und differenzierten Grammatiken werden wegen der hohen Stelligkeit unübersichtlich und fehlerträchtig.

2. Modifikationen an einzelnen Regeln haben u.U. erhebliche Modifikationen der Gesamtgrammatik zur Folge.

3. Einzelne Regeln enthalten Redundanzen und sind deshalb in geringerem Maße explizit: Sie enthalten z.B. anonyme Variablen an den Merkmalspositionen, die für die jeweilige Regel irrelevant sind.

4. Die Schnittstellen zu anderen Komponenten (z.B. zum Lexikon) müssen an die Termstruktur der jeweils verwendeten DCG angepaßt werden. Modifikationen der Grammatik oder des Lexikons haben dann Seiteneffekte auf die jeweils andere Komponente, die eine neue Spezifikation der Schnittstelle zur Folge haben können.

Eine Möglichkeit, die Vorteile von DCGs ohne die inhärenten Probleme, die mit der Beschränkung auf Termunifikation verbunden sind, ausnutzen zu können, besteht darin, die Merkmalsverarbeitung auszugliedern und in *extra-predicates* zu verlagern, die Baum- oder Graphunifikation verwenden. Auf diese Weise lassen sich auch andere Formalismen wie z.B. GPSG und HPSG als DCGs implementieren (vgl. [**?**], [**?**]).
Ein weiterer Kritikpunkt, der in Diskussionen über die Adäquatheit von DCGs immer wieder genannt wird, ist das Problem, daß linksrekursive Regeln nicht terminieren, wenn die Standard-Parsingstrategie für DCGs (Top-down-depth-first-Suche

[1]Zwei Prolog-Terme T_1, T_2 unifizieren nur dann, wenn sie denselben Funktor und dieselbe Stelligkeit besitzen und für alle ihre Argumente gilt: Das i-te Argument von T_1 unifiziert mit dem i-tem Argument von T_2.

mit Backtracking) verwendet wird. Diese Kritik betrifft jedoch nicht DCGs als einen deklarativen Gramatikformalismus, sondern die Verarbeitungsstrategie des verwendeten Parsers. In Kapitel 5 haben wir die Prolog-Implementierung eines Left-corner-Parsers für kontextfreie PSGs vorgestellt, der sich auch für DCGs verwenden läßt. Der folgende Abschnitt beschreibt eine Prolog-Implementierung eines earley-basierten Parsers für DCGs.

12.5 Ein earley-basierter Parser für DCGs

Die Implementierung basiert auf der Implementierung eines Earley-Recognizers in C-Prolog von [**?**]. Die wichtigsten Erweiterungen dieses Systems sind:

- inkrementeller Strukturaufbau (kein Recognizer, sondern Parser)
- auch als Parser für DCGs verwendbar
- Bildschirmdarstellung von Bäumen in lesbarer Grafik (auch für DCGs)

Operatorendefinitionen und Hilfsprädikate:

```
:- op(1100, xfx, '—>').

retractall(X) :-
      retract(X), fail.
retractall(_).

islist([ ]).
islist([_| _]).
```

```
:- op(700, xfy, '\—').

copy_term(X, Y) :-
      retractall(term_copy(_)),
      assert(term_copy(X)),
      term_copy(Y).
```

Hauptprädikate

Für den DCG-Parser benötigen wir folgende Prädikate: parse/1, parse_recursive/2, shift/3, closure/5, neue_kante/5, complete/4, expand/2, skanten/5, finde_alle_skanten/4, finde_skante/4, auswerten/1 regel/2 und lexikon_eintrag/2.

```
% parse/1
% Mit diesem Prädikat wird die Analyse eines Satzes gestartet. Der Satz wird durch
% eine Liste von Prolog-Atomen repräsentiert.
parse(_) :-
    abolish(kante/5),              % „Aufräumen“ der Datenbasis.
    abolish(skante/3),
    start_symbol(S),
    (S —> RHS),                    % Initialisierung der Chart.
    closure(0, 0, S, [ ], RHS),    % Hüllenbildung.
```

```
    fail.                          % Sofern mehrere Startsymbole und/oder
                                   % Startregeln definiert sind: Backtracking.
parse(Satz) :-
    parse_recursive(0, Satz).      % Initialisierung abgeschlossen
% parse_recursive/2
parse_recursive(Ende, [ ]) :-      % Satz ist vollständig abgearbeitet.
    auswerten(Ende).
parse_recursive(Start, [Word | Rest]) :-
    Ende is Start + 1,             % Inkrementierung des Index.
    shift(Start, Ende, Word),      % Aufruf der Shift-Prozedur.
    parse_recursive(Ende, Rest).   % Rekursion mit Restsatz.

% shift/3
% Einlesen der Lexikoneinträge für Word und Aufruf von CLOSURE für jeden ge-
% fundenen Eintrag.
shift(Start, Ende, Word) :-
    lexikon_eintrag(Word, Category),    % Lexikonzugriff.
    assertz(skante(Start, Ende, [Category, [Word]])),
    closure(Start, Ende, Category, [Word], [ ]), fail.
shift(_, _, _).

% closure/5
% Hüllenbildung durch Vorwärtsverkettung.
% Verarbeitung einer passiven Kante: Aufruf von complete/4.
closure(Start, Ende, LHS, Geschlossen, [ ]) :-
    neue_kante(Start, Ende, LHS, Geschlossen, [ ]),
    complete(Start, Ende, LHS, Geschlossen).
closure(Start, Ende, LHS, Geschlossen, [Next | Rest]) :-
    neue_kante(Start, Ende, LHS, Geschlossen, [Next | Rest]),
    expand(Ende, Next).

% neue_kante/5
% Es wird geprüft, ob die Kante von einer anderen Kante in der Chart subsumiert
% wird. Ist das nicht der Fall, wird sie in die Chart eingetragen.
neue_kante(Start, Ende, LHS, Geschlossen, Offen) :-
    copy_term(kante(Start, Ende, LHS, Geschlossen, Offen), KANTE),
    call(KANTE),
    KANTE =.. [_, _, LHS2, _, _ ],
    subsumes(LHS2, LHS), !, fail.
neue_kante(Start, Ende, LHS, Geschlossen, Offen) :-
    assertz(kante(Start, Ende, LHS, Geschlossen, Offen)).
```

```
% complete/4
complete(Start, Ende, B, Cats) :-
    kante(Start0, Start, LHS, Geschlossen0, [B | Offen]),
    append(Geschlossen0, [B], Geschlossen),
    finde_alle_skanten(Start, Ende, B, Cats),
    closure(Start0, Ende, LHS, Geschlossen, Offen), fail.
    complete(_, _, _, _).
% expand/2
expand(Node, Next) :-
    (Next —> RHS),
    closure(Node, Node, Next, [ ], RHS),
    fail.
expand(_, _).
```

Verwaltung von Strukturkanten:

```
% skanten/5
skanten(Start, Ende, LHS, [D1 | Ds], [D1, D1s | X]) :-
    finde_skante(Start, Ende1, D1, D1s),
    skanten(Ende1, Ende, LHS, Ds, X).
    skanten(X, X, _, [ ], [ ]).
% finde_alle_skanten/4
finde_alle_skanten(Start, Ende, LHS, [D1 | Ds]) :-
    skanten(Start, Ende, LHS, [D1 | Ds], X),
    not(skante(Start, Ende, [LHS, X])),
    assertz(skante(Start, Ende, [LHS, X])), fail.
    finde_alle_skanten(_, _, _, _).
% finde_skante/4
finde_skante(Start, Ende, C1, TREE) :-
    skante(Start, Ende, [C1, TREE]).
% auswerten/1
auswerten2(Ende) :-
    start_symbol(S),
    kante(0, Ende, S, _, [ ]),
    printstructures(0, E).
% regel/2
regel(LHS, RHS) :-
    (LHS —> X),
    X =.. RHS.
```

Wir verwenden in dem DCG-Parser eine sehr einfache Definition des Begriffs *lexikalische Kategorie*, die an die jeweils verwendeten Grammatiken und Lexika angepaßt werden kann:

```
% lexikon_eintrag/2
% W ist eine terminale Kategorie gdw. es als einziges Element auf der rechten
% Regelseite einer Regel vorkommt und es keine Regel gibt, die W expandiert.
lexikon_eintrag(W, C) :-
    (C —> [W]),
    not(regel(W, _)).
```

Termunifikation

Die folgende Definition von Term-Subsumtion beruht auf folgendem Prinzip:

1. T_1 subsumiert T_2, wenn $T_1 == T_2$;
2. T_1 subsumiert T_2, wenn die folgenden Bedingungen erfüllt sind:
   (a) T_1 und T_2 unifizieren zu T_3
   (b) die Substitution aller Variablen in T_3 durch Atome der Form var1, var2, usw. führt zu einem Term T_3', der identisch ist mit dem Term T_2', der dadurch entsteht, daß alle Variablen in T_2 in derselben Reihenfolge durch die Atome var1, var2 etc. ersetzt werden.

```
% var_bind/2 ersetzt alle Variablen in einer Liste oder in einem Term durch Atome
% der Form varX (wobei X für eine natürliche Zahl steht).
var_bind(A, B) :-
    set_counter(0),
    copy_term(A, X),
    var_bind2(X, B).

% var_bind2/2
var_bind2(X, X) :-                  % Variable
    var(X), !, gensym(X).
var_bind2([ ], [ ]) :- !.           % Leere Liste
var_bind2(A, A):- atomic(A), !.     % Atom, Ziffer
var_bind2([X | R], [XN | RN]) :-    % Liste
    var_bind2(X, XN),
    var_bind2(R, RN), !.
var_bind2(Term, TermN) :-           % Term
    Term =.. ALIST, !,
    var_bind2(ALIST, ALISTN),
    TermN =.. ALISTN, !.
```

```
% gensym/1
% Das Prädikat erzeugt ein Atom der Form varX (X ist eine Ziffer).
gensym(X) :-
    current_counter(NUM), !,
    retract(current_counter(NUM)),
    NEW is NUM + 1,
    assertz(current_counter(NEW)),
    name(NEW, NEWLIST),
    append([118, 97, 114], NEWLIST, XLIST),
    name(X, XLIST).
gensym(var0) :-
    assertz(current_counter(0)), !.

% set_counter/1
set_counter(X) :-
    retractall(current_counter(_)),
    assertz(current_counter(X)).

% subsumes/2
% Das Prädikat testet, ob Term A den Term B subsumiert.
subsumes(A, B) :- A == B, !.[2]
subsumes(A, B) :-
    unify(A, B, C),
    var_bind(B, B1),
    var_bind(C, C1),
    B1 == C1.

% unify/3
% Das Prädikat unifiziert Kopien der Terme A und B und gibt als Resultat deren
% Unifikation C aus.
unify(A, B, A1) :-
    copy_term(A, A1),
    copy_term(B, B1),
    A1 = B1.
```

Semi-graphische Strukturausgabe

```
% structures/0
% Das Prädikat zeigt alle von COMPLETE entdeckten Teilbäume.
```

[2] Der Test auf Identität ist logisch nicht erforderlich, weil der Fall der Identität auch durch die zweite Definition erfaßt wird. Da Identität jedoch relativ häufig vorkommt, wird aus Effizienzgründen zunächst mit % "==" geprüft.

```
structures :-
    skante(_, _, C),
    treeprint(C, ''), nl, nl, nl, get0_noecho(_),
    fail.
structures.

% printstructures/2
printstructures(F, T) :-
    skante(F, T, X),
    treeprint(X, ''), nl, nl, nl, fail.
printstructures(_, _).
```

Die folgenden Prädikate wandeln Terme in Atome um. Der Zweck dieser Umwandlung ist es, die Bildschirmdarstellung mit treeprint/2 zu ermöglichen: pred2atom/2 bildet einen Term $p(a_1, \ldots, a_n)$ auf p ab; d.h. es reduziert für die Bildschirmdarstellung durch treeprint/2 einen komplexen DCG-Term auf ein Atom.

```
% pred2atom
pred2atom(TERM, ATOM) :-
    TERM =.. [ATOM | _].
pred2atom(NUM, ATOM) :-
    number(NUM),
    name(NUM, NAME),
    name(ATOM, NAME).

% showitems/0
% Das Prädikat zeigt alle vollständigen Kanten in Baumdarstellung.
showitems :-
    skante(_, _, D),
    treeprint(D, ''), nl, nl, nl, get0_noecho(101).
```

Es folgen die Prädikate für den Pretty-Printer für Bäume.

```
% treeprint/2
% Hauptprädikat des Pretty-Printers.
treeprint([M, [T]], TAB) :-                    % Lexikalische Kategorie.
    nolist(M), nolist(T),
    pred2atom(M, M2), pred2atom(T, T2),
    !, write(M2), write('—'), write(T2), nl, !.
treeprint([M, [T1, T2 | R]], TAB) :-           % Linke Tochter.
    nolist(M), nolist(T1), nolist(T2), !,      % Lexikalische Kategorie.
    pred2atom(M, M2), pred2atom(T1, T12),
    pred2atom(T2, T22),
    write(M2), write('—'), write(T12), nl, !,  % Weitere Töchter.
```

```
    mehrzweige([T22 | R], TAB), !.
treeprint([Mutter, [A, B | R]], TAB) :-        % Erste Verzweigung.
    !, pred2atom(Mutter, Mutter2),
    write(Mutter2), write('—'),
    name(Mutter2, MN), length(MN, ML2),
    ML3 is ML2 + 2, makestr(ML3, MS), !,
    ((R \= [ ], !, strappend('|', MS, MS2));
     (strappend(' ', MS, MS2))),
    strappend(TAB, MS2, NTAB), !,
    treeprint([A, B], NTAB), !,
    mehrzweige(R, TAB), !.
% mehrzweige/2
mehrzweige([A, B, C, D | R], TAB) :- % Eine Verzweigung.
    nolist(A), islist(B),              % Syntaktische Kategorie.
    !, write(TAB),
    write('\—'),
    strappend(TAB, '| ', NT), !,
    treeprint([A, B], NT), !,
    mehrzweige([C, D | R], TAB), !.
mehrzweige([A, B | R], TAB) :-         % Letzte Verzweigung.
    nolist(A), islist(B), !,           % Syntaktische Kategorie.
    write(TAB),
    write('\—'),
    strappend(TAB, ' ', NT),
    treeprint([A, B], NT),
    mehrzweige(R, TAB), !.
mehrzweige([A],TAB) :-                 % Letzte Verzweigung.
    nolist(A), !,                      % Lexikalische Kategorie.
    write(TAB),
    write('\—'),
    pred2atom(A, A2),
    write(A2), nl, !.
mehrzweige([A, B | R], TAB) :-         % Verzweigung lex. Kategorien.
    nolist(A), nolist(B),
    !, write(TAB),
    write('\—'),
    pred2atom(A, A2),
    write(A2), nl,
    mehrzweige([B | R], TAB), !.
mehrzweige([ ],_) :- !.
```

Hilfsprädikate für den Pretty-Printer:

```
% nolist/1
nolist(X) :- not(islist(X)).
```

```
% strappend/3
strappend('', X, X) :- !.
strappend(A, B, C) :-
    !, name(A, A2),
    name(B, B2),
    append(A2, B2, C2),
    name(C, C2), !.
```

```
% makestr/3
makestr(- 1, _) :- !, fail.
makestr(0, '') :- !.
makestr(X, Y) :-
    !, X2 is X - 1,
    makestr(X2, Y2),
    strappend(Y2, ' ', Y), !.
```

Aufgaben

12.1 Schreiben Sie jeweils eine DCG für die Sprachen:

(a) $a^n\ b^m\ c^n\ d^m$ und

(b) $a^n\ b^n\ c^n$.

12.2 In [**?**] wird eine lexikalistische Lösung für die Beschreibung von Komplementstrukturen in Verbalphrasen gegeben. Statt verschiedener VP-expandierender Regeln für Verben mit unterschiedlichen Subkategorisierungsrahmen wie unter (a) gibt Shieber nur genau eine VP-Regel im PATR-II-Formalismus an (b), durch die die Komplemente des Verbs instantiiert werden:

(a) VP → V[lesen] NP
VP → V[stellen] NP PP
VP → V[schlafen]
VP → V[behaupten] S
...

(b) VP → V[subcat: X] X

Schreiben Sie eine DCG, die nach diesem Prinzip einige deutsche Verben (z.B. lesen, stellen, schlafen) behandelt. Diese DCG sollte Sätze wie die folgenden beschreiben können:

sie schläft
sie liest ein buch
sie legt ein buch auf ein buch

Die Subkategorisierungsinformationen für die einzelnen Verben sollten dabei nur in den Lexikoneinträgen, nicht jedoch in den Regeln, festgelegt werden.

Kapitel 13

Ein Chart-Parser für unifikationsbasierte Syntaxen

Grundsätzlich lassen sich alle der von uns vorgestellten Parsingalgorithmen so erweitern, daß sie zum Parsen unifikationsbasierter Syntaxen verwendet werden können, die einen *kontextfreien Kern* enthalten. Wir werden anhand des Earley-Algorithmus die Schritte skizzieren, die ein solcher Adaptionsprozeß erfordert. Wir beschränken uns auf die Implementierung eines earley-basierten Recognizers in Lisp.

13.1 Indirektes Parsen

Ähnlich wie bei den ID/LP-Syntaxen (vgl. Kapitel 9) lassen sich auch bei den unifikationsbasierten Syntaxen zwei unterschiedliche Strategien verfolgen, die wir im folgenden als „direktes“ bzw. „indirektes“ Parsen bezeichnen werden. Charakteristisch für das indirekte Parsen mit unifikationsbasierten Syntaxen ist, daß die Syntax entweder vollständig in eine einfache kontextfreie Syntax kompiliert wird oder zum Parsen nur der kontextfreie Kern der Syntax verwendet wird.

Eine unifikationsbasierte Syntax G kann unter bestimmten Voraussetzungen als eine komprimierte kontextfreie Syntax betrachtet werden; d.h. jede Regel wird als „Abkürzung“ für eine Menge kontextfreier Regeln interpretiert.

Beispiel (13-1)

Wenn man voraussetzt, daß das Merkmal *agreement* als Wert eine Merkmalsstruktur nimmt, in der die Merkmale *gen*[3], *kas*[4], *num*[2][1] spezifiziert sind, und die Merkmale vernachlässigt, die außerdem noch als *head*-Merkmale von Nomen auftreten können, dann kann die folgende Regel

$$\left[\begin{array}{ll} cat & NP \\ head & \boxed{1}\left[\begin{array}{ll} agreement & \boxed{2} \end{array}\right] \end{array}\right] \longrightarrow \left[\begin{array}{ll} cat & det \\ agreement & \boxed{2} \end{array}\right] \left[\begin{array}{ll} cat & n \\ head & \boxed{1} \end{array}\right]$$

als Zusammenfassung von 24 kontextfreien Regeln aufgefaßt werden:

[1] Die in Klammern angegebene Ziffer gibt die Anzahl der verschiedenen Werte dieses Merkmals an.

$$
\begin{array}{lll}
NP_{num(1)gen(1)kas(1)} \rightarrow det_{num(1)gen(1)kas(1)} & n_{num(1)gen(1)kas(1)} \\
NP_{num(1)gen(2)kas(1)} \rightarrow det_{num(1)gen(2)kas(1)} & n_{num(1)gen(2)kas(1)} \\
\vdots & \vdots & \vdots \\
NP_{num(2)gen(3)kas(4)} \rightarrow det_{num(2)gen(3)kas(4)} & n_{num(2)gen(3)kas(4)}
\end{array}
$$

Zum Parsen wird G in eine einfache kontextfreie Syntax G' kompiliert, für die dann ein beliebiger, für kontextfreie Syntaxen geeigneter Parsingalgorithmus verwendet werden kann. Der entscheidende Nachteil dieses Verfahrens liegt in der kombinatorischen Explosion, zu der die Expandierung der unifikationsbasierten Quellsyntax in eine kontextfreie Objektsyntax führt. Angenommen, zur Bildung von Kategorien stehen 20 binäre Merkmale zur Verfügung, dann gibt es unter der Voraussetzung, daß alle Merkmale in jeder Kategorie vorkommen können

- $2^{20} = 1048576$ Kategorien, sofern jedes Merkmals in jeder Kategorie spezifiziert wird bzw.

- $3^{20} = 3486784401$ Kategorien, wenn Kategorien nicht für alle Merkmale spezifiziert sein müssen.

Da die Größe der Grammatik den Zeit- und Raumbedarf eines Parsers direkt beeinflußt, ist dieses Verfahren nicht praktikabel. Außerdem gibt es unifikationsbasierte Formalismen, in denen – anders als in der GPSG – die Menge der lexikalischen und syntaktischen Kategorien nicht endlich ist:

Beispiel (13-2)
Wenn ein beliebiges Merkmal f als Wert eine Kategorie akzeptiert, in der f selbst wieder spezifiziert sein kann und prinzipiell keine Obergrenze für diese Art der rekursiven Selbsteinbettung gesetzt wird, ist die Menge der auf diese Weise erzeugbaren Kategorien abzählbar unendlich (vgl. [**?**]):

$$
\left[\begin{array}{ll}
\ldots & \\
f & \left[\begin{array}{ll} \ldots & \\ f & [\ldots] \\ \ldots & \end{array}\right] \\
\ldots &
\end{array}\right]
$$

Eine Möglichkeit, diese Probleme zu lösen, die es außerdem noch erlaubt, die Standard-Parsingalgorithmen unmodifiziert auch für die Analyse von unifikationsbasierten Syntaxen zu verwenden, besteht darin, zum Parsen nur den kontextfreien Kern dieser Syntaxen zu verwenden. Dieses Verfahren ist allerdings nur

dann anwendbar, wenn sichergestellt ist, daß alle Merkmalsstrukturen Informationen darüber enthalten, welchen Typ von Kategorie sie repräsentieren. Wenn z.B. das Merkmal *cat* mit den Werten S, NP, $VP, \ldots, det$, n, $v, \ldots$ verwendet wird, um kategoriale Informationen zu kodieren, ist zu fordern, daß dieses Merkmal in jeder Merkmalsstruktur spezifiziert ist. Alle übrigen in den Merkmalsstrukturen spezifizierten Merkmale werden zunächst ignoriert und nach dem Parsen eines Satzes dazu verwendet, die Strukturbeschreibungen auszufiltern, die Merkmalsstrukturen mit inkompatiblen Informationen enthalten.

Der Nachteil dieses typischen „Generate-and-test"-Ansatzes besteht darin, daß eine u.U. nicht unerhebliche Übergenerierung in Kauf genommen wird, da beim Parsen nicht alle verfügbaren Informationen dazu verwendet werden, um zu einem möglichst frühen Zeitpunkt alle Analysen zu verwerfen, die nicht erfolgreich zu Ende geführt werden können. Anders formuliert: Dieses Verfahren ist weder elegant (sequentielle Ausführung koordinierbarer Aktionen) noch effizient (Übergenerierung). Auch für Formalismen wie die LFG, die ein derartiges zweistufiges Verfahren nahezulegen scheint, da sie jedem wohlgeformten Satz eine Konstituentenstruktur und eine Merkmalsstruktur (*funktionale Beschreibung*) zuordnet, sind effiziente Parser entwickelt worden, die beide Strukturen parallel berechnen (vgl. [**?**]).

Beispiel (13-3)

G sei eine einfache Syntax, die u.a. die folgenden Regeln enthält:

Syntaktische Regeln:

$$\begin{bmatrix} cat & S \\ head & \boxed{1}\begin{bmatrix} subject & \boxed{2} \end{bmatrix} \end{bmatrix} \longrightarrow \begin{bmatrix} cat & NP \\ head & \boxed{2} \end{bmatrix} \begin{bmatrix} cat & VP \\ head & \boxed{1}[\ vform\ finite\ ] \end{bmatrix}$$

$$\begin{bmatrix} cat & NP \\ head & \boxed{1}\begin{bmatrix} agreement & \boxed{2} \end{bmatrix} \end{bmatrix} \longrightarrow \begin{bmatrix} cat & det \\ agreement & \boxed{2} \end{bmatrix} \begin{bmatrix} cat & n \\ head & \boxed{1} \end{bmatrix}$$

$$\begin{bmatrix} cat & VP \\ head & \boxed{1} \end{bmatrix} \longrightarrow \begin{bmatrix} cat & v \\ head & \boxed{1} \end{bmatrix}$$

Lexikalische Informationen:

$$\begin{bmatrix} cat & det \\ head & \begin{bmatrix} agreement & \begin{bmatrix} num & sing \\ gen & fem \end{bmatrix} \end{bmatrix} \end{bmatrix} \longrightarrow die_1$$

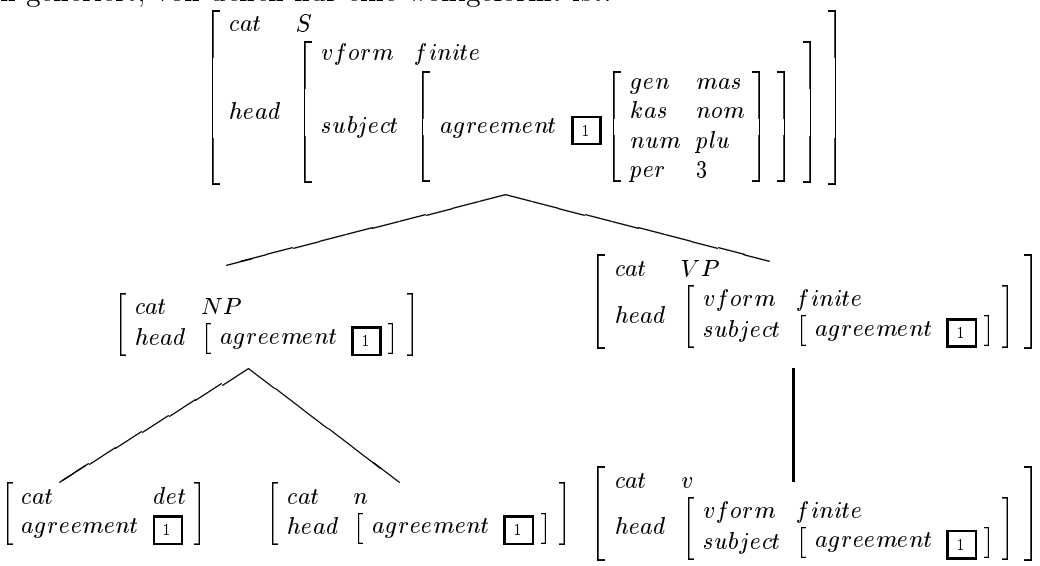
$$\begin{bmatrix} cat & det \\ head & \begin{bmatrix} agreement & \begin{bmatrix} num & plu \\ kas & nom \end{bmatrix} \end{bmatrix} \end{bmatrix} \longrightarrow die_2$$

$$\begin{bmatrix} cat & n \\ head & \begin{bmatrix} agreement & \begin{bmatrix} gen & mas \\ kas & nom \end{bmatrix} \end{bmatrix} \end{bmatrix} \longrightarrow schüler$$

$$\begin{bmatrix} cat & v \\ head & \begin{bmatrix} vform & finite \\ subject & \begin{bmatrix} agreement & \begin{bmatrix} num & plu \\ per & 3 \end{bmatrix} \end{bmatrix} \end{bmatrix} \end{bmatrix} \longrightarrow pfeifen_1$$

$$\begin{bmatrix} cat & v \\ head & \begin{bmatrix} vform & infinite \end{bmatrix} \end{bmatrix} \longrightarrow pfeifen_2$$

Wenn $\alpha = Die\ Schüler\ pfeifen$, dann werden für α $2 * 1 * 2 = 4$ Strukturbeschreibungen generiert, von denen nur eine wohlgeformt ist:



Die Strukturbeschreibungen, die auf dem die_1-Eintrag basieren, fordern verschiedene Werte für das *gen*-Merkmal der Nominalphrase; $pfeifen_2$ führt zu einem *vform*-Konflikt in der Verbalphrase.

Ein weiteres Problem bildet die Forderung, daß jede Merkmalsstruktur kategoriale Informationen enthalten soll. Die Mehrzahl der unifikationsbasierten Formalismen erlaubt die Verwendung stark unterspezifizierter Merkmalsstrukturen, die u.a. zur Beschreibung des Subkategorisierungsrahmens von Verben und zur Behandlung von diskontinuierlichen Konstituenten verwendet werden.

Beispiel (13-4)

(a) PATR-II

$VP_1 \rightarrow VP_2\ X$
$< VP_1\ head > \ = \ < VP_2\ head >$
$< VP_2\ subcat\ first > \ = \ < X >$
$< VP_2\ subcat\ rest > \ = \ < VP_1\ subcat >$ [?, S.33]

Das *Subcat*-Merkmal, durch das der Subkategorisierungsrahmen von Ausdrücken spezifiziert wird, hat die folgende Struktur:

$$subcat \left[\begin{array}{ll} first & Komplement_n \\ rest & \left[\begin{array}{ll} first & Komplement_{n-1} \\ rest & \left[\begin{array}{ll} \ldots & \left[\begin{array}{ll} first & Komplement_1 \\ rest & end \end{array}\right] \end{array}\right] \end{array}\right] \end{array}\right]$$

Die Indizes dienen dazu, die verschiedenen Vorkommen des VP-Symbols zu unterscheiden; X ist eine Variable. Die Regel legt fest, daß eine Verbalphrase (VP_1) aus einer anderen Verbalphrase (VP_2) und der Kategorie (X) bestehen kann, die an erster Position des Subkategorisierungsrahmens von VP_2 verzeichnet ist (*subcat first*), und der Subkategorisierungsrahmen von VP_2 aus den übrigen Kategorien besteht (*subcat rest*).

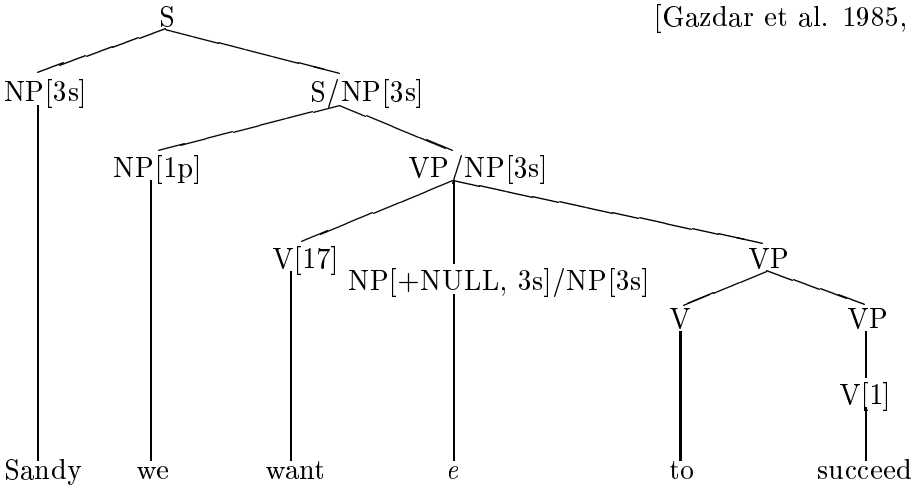
(b) GPSG

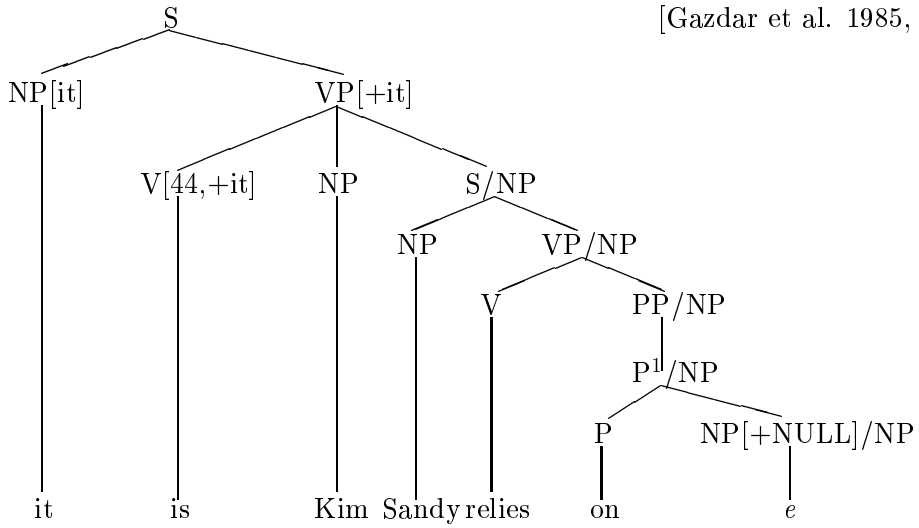
$S \rightarrow X^2,\ H/X^2$

$VP[+it] \rightarrow H[44],\ X^2,\ S[FIN]/X^2$ [**?**, S.248]

Beide Regeln sind motiviert durch (englische) Sätze, in denen die Objekt-NP topikalisiert wird. Sie führen sogenannte *SLASH*-Kategorien ein (H/X^2 bzw. $S[FIN]/X^2$), die dazu verwendet werden, Konstituenten zu repräsentieren, in denen eine Subkonstituente fehlt. „H“ bezeichnet die *head*-Kategorie der Konstituente und „H[44]“ die lexikalische Kategorie, die das Verb *be* enthält. „X^2“ steht für eine beliebige phrasale Kategorie, d.h. eine Kategorie mit dem *Bar-Level* 2, wie z.B. VP, NP, AP, PP und in der GPSG auch S. Die Analysen basieren in beiden Fällen auf der Annahme, daß die Objekt-NP von ihrer Position am Satzende vor die Subjekt-NP 'bewegt' wird und diese Bewegung eine *Spur* (e) zurückläßt, wobei in der GPSG allerdings auf Bewegungstransformationen (wie *Move-*α im GB-Modell) verzichtet wird.

[Gazdar et al. 1985, S.145]





[Gazdar et al. 1985, S.159]

Die Verwendung von Merkmalsstrukturen dieses Typs läßt die Reduzierung einer unifikationsbasierten Syntax auf ihren kontextfreien Kern zu einem nicht-trivialen Problem werden.

13.2 Direktes Parsen

Verzichtet man darauf, eine unifikationsbasierte Syntax in eine kontextfreie Syntax zu expandieren bzw. sie auf ihren kontextfreien Kern zu reduzieren, ist es erforderlich, den Parser so zu modifizieren, daß er die durch Merkmalsstrukturen repräsentierten (komplexen) Symbole korrekt verarbeiten kann. Wir beschreiben zunächst die von uns verwendete Repräsentation für Kategorien und Regeln und gehen dann auf die notwendigen Änderungen des Earley-Algorithmus ein.

13.2.1 Repräsentation von Regeln und Kategorien

Die Wahl einer Repräsentationsform für Kategorien und Regeln wird entscheidend beeinflußt von den Operationen, die auf ihnen zu definieren sind. Die fundamentale Operation auf Kategorien in merkmalsbasierten Grammatikformalismen ist die Unifikation. Es gibt verschiedene Möglichkeiten, diese Operation zu realisieren:

1. *Unifikation als generativer Prozeß*
   Die Unifikation zweier Merkmalsstrukturen M_1, M_2, kann als Verfahren aufgefaßt werden, das zur Generierung einer neuen Merkmalsstruktur M_3 führt: sukzessiv werden M_1 und M_2 abgearbeitet und dabei alle notwendigen Informationen in die neue Merkmalsstruktur M_3 kopiert. Der Nachteil dieses Vorgehens liegt darin, daß die schrittweise Erzeugung der neuen Merkmalss-

truktur relativ aufwendig ist und dieser Aufwand besonders dann als unverhältnismäßig erscheint, wenn sich M_1 und M_2 nur geringfügig unterscheiden.

2. *Destruktive Unifikation*
   Statt eine neue Merkmalsstruktur zu erzeugen, kann eine der beiden Merkmalsstrukturen so modifiziert werden, daß sie das Ergebnis der Unifikation beider Merkmalsstrukturen repräsentiert; d.h. $M_1 \sqcup M_2 = M_1'$ bzw. $M_1 \sqcup M_2 = M_2'$.
   Destruktive Realisierungen von Unifikationsalgorithmen sind in der Regel deutlich effizienter als nicht-destruktive Varianten. Allerdings sind destruktive Operationen auf Datenstrukturen grundsätzlich problematisch, da sie leicht nicht-intendierte Seiteneffekte mit ungeahnten Konsequenzen verursachen können, und außerdem geht eine der urspünglichen Merkmalsstrukturen verloren.

Eine Lösung, die häufig gewählt wird und die auch wir im folgenden verwenden werden, besteht darin, eine *Kopie* von M_1 bzw. M_2 so zu erweitern, daß sie *implizit* das Ergebnis der Unifikation von M_1 und M_2 repräsentiert[2]. Wir repräsentieren eine Merkmalsstruktur M_i als ein geordnetes Paar <*Pfadmenge, Bindungsmenge*>, mit *Pfadmenge* als die Menge aller Pfade von M_i und *Bindungsmenge* als die Menge aller Bindungen der Variablen, die in *Pfadmenge* vorkommen. Wir verwenden folgende Syntax:

Merkmalsstruktur := <Pfadmenge, Bindungsmenge>
Pfadmenge := $\{Pfad_1, \ldots, Pfad_n\}$
Bindungsmenge := $\{Bindung_1, \ldots, Bindung_m\}$
Pfad := <Attribut, Wert>
Attribut := Symbol
Wert := Symbol|Variable|$Pfadmenge$
Bindung := <Variable, Wert>

Ausgehend von dieser Repräsentation liegt es nahe, eine Regel r der Form $X \rightarrow Y_1 \ldots Y_p$ $(p \geq 1)$ einfach als eine Folge von Kategorien, d.h. geordneten Paaren der gerade spezifizierten Form, darzustellen:

$$<<PM_X, BM_X>, <PM_{Y_1}, BM_{Y_1}>, \ldots, <PM_{Y_p}, BM_{Y_p}>>$$

Da aber in Regeln Variablen häufig verwendet werden um festzulegen, daß bestimmte Merkmale in verschiedenen Kategorien denselben Wert aufweisen müssen, fassen wir die Variablenbindungen aller Kategorien zusammen:

[2]Eine andere, interessante Variante wird in [?] vorgestellt: Ihr Unifikationsalgorithmus liefert als Resultat eine *Substitution*, d.h. eine Menge von <*Variable*, *Wert*> Paaren, durch die sowohl M_1 und M_2 in eine Merkmalsstruktur überführt werden können, die die Unifikation beider Strukturen repräsentiert.

$$<<PM_X,\ PM_{Y_1},\ldots,\ PM_{Y_p}>,\ BM_r>$$[3]

[3] In den folgenden Beispielen verwenden wir allerdings weiterhin die üblichen Attribut-Wert-Matrizen.

Beispiel (13-5)
Die folgenden drei Beispiele zeigen, wie sich die Repräsentation von Kategorien bzw. Regeln in unserer Notation von der Attribut-Wert-Matrizen-Notation unterscheidet:

(a) die leere Kategorie:
[ ]
(() ())

(b) eine Nominalphrase mit den Merkmalen Kasus = Nominativ und Person = 3:

$$\begin{bmatrix} cat & NP \\ head & \begin{bmatrix} per & 3 \\ kas & nom \end{bmatrix} \end{bmatrix}$$

(((cat NP) (head ((per 3) (kas nom)))) ())

(c) eine Regel S → NP VP:

$$\begin{bmatrix} cat & S \\ head & \boxed{1}\begin{bmatrix} agr & \boxed{2} \end{bmatrix} \end{bmatrix} \rightarrow \begin{bmatrix} cat & NP \\ head & \boxed{2} \end{bmatrix} \begin{bmatrix} cat & VP \\ head & \boxed{1}\begin{bmatrix} agr & \boxed{2} \end{bmatrix} \end{bmatrix}$$

((((cat S) (head _X1)) ((cat NP) (head _X2)) ((cat VP) (head _X1)))
((_X1 ((agr _X2)))))

Die Werte der *head*-Merkmale von S und VP müssen übereinstimmen und ebenso der Wert des *agr*(*eement*)-Merkmals von VP (und S) und der des *head*-Merkmals von NP (NP-VP-Kongruenz).

13.2.2 Parsen mit merkmalsbasierten Syntaxen

Es sind nur relativ geringe Änderungen erforderlich, um den Earley-Algorithmus zur Verarbeitung von Syntaxen mit komplexen Kategoriensymbolen verwenden zu können[4]:

Kanten. In den Kanten müssen neben den üblichen Angaben auch Informationen über die Variablenbindungen, die innerhalb der in ihnen vorkommenden Kategorien (DAGs) bestehen, gespeichert werden. Wie auch bei der Repräsentation der Regeln fassen wir alle Bindungen in einer Bindungsliste (**variablen**) zusammen; d.h., Kanten haben das folgende Format:

[*start, ende*, *kopf*, *geschlossen*, *offen*, **variablen**]

[4]Wir legen die Version des Earley-Algorithmus zugrunde, bei der die Chart durch *Vorwärtsverkettung* berechnet wird; vgl. RECOGNIZE-EARLEY-2, Kapitel 7.2.1.

Berechnung der Chart. Die zur Berechnung der Chart notwendigen Operationen EXPAND, COMPLETE und SHIFT müssen an diese veränderte Kantenstruktur angepaßt werden. Außerdem ist in ihnen jede Bedingung, in der die Identität zweier Kategorien K_1, K_2 gefordert wird ($K_1 = K_2$), zu ersetzen durch die Bedingung, daß beide Kategorien unifizieren ($K_1 \sqcup K_2$). Wir erhalten so:

PROZEDUR EXPAND$_{DAG}$

DATEN: Eine kontextfreie Syntax G = <V_N, V_T, S, R> und eine Chart C.
EINGABE: Eine Kante k = [i, j, A, α, Bβ, θ].

METHODE:
Für alle B' $\rightarrow \tau \in$ R:
Wenn < B $\sqcup$ B'>
Dann <*CLOSURE*([j, j, B*, e, τ, θ'][5])>

PROZEDUR COMPLETE$_{DAG}$

DATEN: Eine Chart C.
EINGABE: Eine Kante k = [j, l, B, τ, e, θ_1].

METHODE:
Für alle [i, j, A, α, B'β, θ_2] $\in$ C, mit $0 \leq i \leq j \leq l \leq n$:
Wenn < B $\sqcup$ B' >
Dann <*CLOSURE*([i, l, A, αB*, β , $\theta_1 \sqcup \theta_2$])>

PROZEDUR SHIFT$_{DAG}$

DATEN: Ein Lexikon L.
EINGABE: Ein Symbol des zu analysierenden Satzes w_j, mit $1 \leq j \leq n$.

METHODE:
Für alle lexikalischen Kategorien B, mit $w_j \in$ B:
<*CLOSURE*([j-1, j, B, w_j, e, θ])>

Die weitreichendsten Änderungen sind bei der Prozedur CLOSURE erforderlich, die die eigentliche Hüllenbildung realisiert; denn die Verwendung von Merkmalsstrukturen anstelle einfacher Kategoriensymbole führt zu zwei Problemen:

1. Die Frage, ob eine Kante neue Informationen repräsentiert, die noch nicht in der Chart enthalten sind, läßt sich nicht mehr so einfach beantworten wie

[5] B* ist die Kategorie, die aus der Unifikation von B und B' resultiert; θ' die durch die Unifikation beider Kategorien möglicherweise modifizierte Menge von Variablenbindungen. Bei syntaktischen Kanten besteht α bzw. β nicht mehr aus einfachen (Kategorien-)Symbolen, sondern aus Merkmalsstrukturen.

bisher.

2. Verschiedene Kanten dürfen nicht Variablen enthalten, deren Namen identisch ist. Anderenfalls ist es möglich, daß die Unifikation von Kategorien zweier Kanten (z.B. beim Kombinieren dieser Kanten) zu fehlerhaften Ergebnissen führt.

Für das zweite Problem gibt es eine naheliegende, wenn auch nicht besonders effiziente Lösung: Um sicherzustellen, daß verschiedene Kanten keine gemeinsamen Variablen enthalten, werden Kanten nicht direkt in die Chart eingetragen, sondern erst nachdem die in ihnen vorkommenden Variablen durch neue Variablen ersetzt wurden (Prozedur KOPIERE-KANTE s.u.).

Beispiel (13-6)
G sei eine merkmalsbasierte Syntax, die u.a. die folgenden Regeln enthält:

$$\begin{bmatrix} cat & S \\ head & \boxed{1} \end{bmatrix} \rightarrow \begin{bmatrix} cat & NP \\ head & \boxed{2} \end{bmatrix} \begin{bmatrix} cat & VP \\ head & \boxed{1}\begin{bmatrix} agr & \boxed{2} \end{bmatrix} \end{bmatrix}$$

$$\begin{bmatrix} cat & VP \\ head & \boxed{3} \end{bmatrix} \rightarrow \begin{bmatrix} cat & v \\ head & \boxed{3} \end{bmatrix} \begin{bmatrix} cat & NP \end{bmatrix}$$

$$\begin{bmatrix} cat & NP \\ head & \boxed{4} \end{bmatrix} \rightarrow \begin{bmatrix} cat & n \\ head & \boxed{4} \end{bmatrix}$$

$$\begin{bmatrix} cat & NP \\ head & \boxed{5} \end{bmatrix} \rightarrow \begin{bmatrix} cat & det \end{bmatrix} \begin{bmatrix} cat & n \\ head & \boxed{5} \end{bmatrix}$$

(a) Wird die Chart mit der Kante

$$[0,\ 0,\ \begin{bmatrix} cat & S \\ head & \boxed{14} \end{bmatrix},\ e,\ \begin{bmatrix} cat & NP \\ head & \boxed{15} \end{bmatrix} \begin{bmatrix} cat & VP \\ head & \boxed{14} \end{bmatrix},\ \left\{ \begin{bmatrix} \boxed{14} & \begin{bmatrix} agr & \boxed{15} \end{bmatrix} \end{bmatrix} \right\}]$$

initialisiert, erzeugt EXPAND_{DAG} die Kanten:

$$[0,\ 0,\ \begin{bmatrix} cat & NP \\ head & \boxed{16} \end{bmatrix},\ e,\ \begin{bmatrix} cat & n \\ head & \boxed{16} \end{bmatrix},\ \{\ \}] \text{ und}$$

$$[0,\ 0,\ \begin{bmatrix} cat & NP \\ head & \boxed{17} \end{bmatrix},\ e,\ \begin{bmatrix} cat & det \end{bmatrix} \begin{bmatrix} cat & n \\ head & \boxed{17} \end{bmatrix},\ \{\ \}]$$

(b) Wenn die Chart die Kanten

$$[2,\ 2,\ \begin{bmatrix} cat & VP \\ head & \boxed{22} \end{bmatrix},\ e,\ \begin{bmatrix} cat & v \\ head & \boxed{22} \end{bmatrix} \begin{bmatrix} cat & NP \end{bmatrix},\ \{\ \}] \text{ und}$$

$$[2,\ 3,\ \begin{bmatrix} cat & v \\ head & \begin{bmatrix} per & 3 \\ tmp & pr\ddot{a}s \\ num & sg \end{bmatrix} \end{bmatrix},\ \text{schläft},\ e,\ \{\ \}]$$

enthält, dann erzeugt COMPLETE$_{DAG}$ die Kante:

$$[2,\ 3,\ \begin{bmatrix} cat & VP \\ head & \boxed{23} \end{bmatrix},\ \begin{bmatrix} cat & v \\ head & \boxed{23} \end{bmatrix},\ \begin{bmatrix} cat & NP \end{bmatrix},\ \left\{ \begin{bmatrix} \boxed{23} & \begin{bmatrix} per & 3 \\ tmp & präs \\ num & sg \end{bmatrix} \end{bmatrix} \right\}]$$

Das zur Lösung des ersten Problems übliche Verfahren besteht darin zu fordern, daß eine Kante k nur dann in die Chart eingetragen wird, wenn es keine Kante k' $\in$ Chart gibt, die k *subsumiert*.

Definition (13-1) *Kantensubsumtion*

Eine Kante k' subsumiert eine Kante k (k' $\preceq$ k) gdw. gilt:

(i). *start*(k') = *start*(k)

(ii). *ende*(k') = *ende*(k)

(iii). *kopf*(k') $\preceq$ *kopf*(k)

(iv). *offen*(k') = x_1' ... x_n', *offen*(k) = x_1 ... x_n ($0 \leq n$) und für alle x_i, $x_{i'}$ ($1 \leq i \leq n$) gilt: x_i' $\preceq$ x_i

(v). *geschlossen*(k') = x_1' ... x_m', *geschlossen*(k) = x_1 ... x_m ($0 \leq m$) und für alle x_j, $x_{j'}$ ($1 \leq j \leq m$) gilt: x_j' $\preceq$ x_j

Durch die Forderung, daß es keine Kante geben darf, die die einzutragende Kante subsumiert, wird ausgeschlossen, daß Kanten eingetragen werden, die sich nur dadurch von bereits eingetragenen Kanten unterscheiden, daß sie zusätzliche Informationen bzw. Merkmalsspezifikationen enthalten.

Beispiel (13-7)

Betrachten wir die folgende, sehr einfache Syntax[6]:

$$\begin{bmatrix} cat & S \end{bmatrix} \rightarrow \begin{bmatrix} cat & X \\ f & \boxed{1} \end{bmatrix}$$

$$\begin{bmatrix} cat & X \\ f & \boxed{2} \end{bmatrix} \rightarrow \begin{bmatrix} cat & X \\ f & \begin{bmatrix} f & \boxed{2} \end{bmatrix} \end{bmatrix} \begin{bmatrix} cat & Y \end{bmatrix}$$

$$\begin{bmatrix} cat & X \end{bmatrix} \rightarrow \begin{bmatrix} cat & Y \end{bmatrix}$$

Die Regel S $\rightarrow$ X führt zu Generierung der Kante

$$k_1 = [0,\ 0,\ \begin{bmatrix} cat & S \end{bmatrix},\ e,\ \begin{bmatrix} cat & X \\ f & \boxed{11} \end{bmatrix},\ \{\ \}].$$

[6] Frei nach [?]

Im nächsten Schritt generiert EXPAND$_{DAG}$ die Kante

$$k_2 = [0, 0, \left[\begin{array}{ll} cat & X \\ f & \boxed{12} \end{array}\right], e, \left[\begin{array}{lll} cat & X & \\ f & \left[\begin{array}{ll} f & \boxed{12} \end{array}\right] & \end{array}\right] \left[\begin{array}{ll} cat & Y \end{array}\right], \{\ }].$$

die auch in die Chart eingetragen wird. Sie aktiviert erneut EXPAND$_{DAG}$:

$$k_3 = [0, 0, \left[\begin{array}{ll} cat & X \\ f & \boxed{13} \end{array}\right], e, \left[\begin{array}{lll} cat & X & \\ f & \left[\begin{array}{ll} f & \boxed{13} \end{array}\right] & \end{array}\right] \left[\begin{array}{ll} cat & Y \end{array}\right], \left\{\left[\begin{array}{ll} \boxed{13} & \left[\begin{array}{ll} f & \boxed{14} \end{array}\right] \end{array}\right]\right\}].$$

Da $k_2 \preceq k_3$, wird k_3 nicht in die Chart eingetragen. Ohne die Subsumtions-Bedingung dagegen würde EXPAND$_{DAG}$ eine infinite Folge von Kanten generieren und die Berechnung der Chart nicht terminieren.

PROZEDUR CLOSURE$_{DAG}$

DATEN: Eine kontextfreie Syntax G, ein Lexikon L und eine Chart C.
EINGABE: Eine aktive/passive Kante k = [i, j, A, α, β, θ].

SEITENEFFEKT: Berechnung der Chart C.

METHODE:
Wenn <k wird von keiner Kante k' $\in$ C subsumiert>
Dann <K $\Leftarrow$ *KOPIERE-KANTE*(k)>
<Trage K in C ein>
Wenn <K ist eine *passive* Kante>
Dann <*COMPLETE$_{DAG}$*(K)>
Sonst <*EXPAND$_{DAG}$*(K)>

Leider stellt die Subsumtionsbedingung nicht sicher, daß die Berechnung der Chart in jedem Fall terminiert.

Beispiel (13-8)
Wenn wir in der Syntax des letzten Beispiel die erste Regel ersetzen durch:

$\left[\begin{array}{ll} cat & S \end{array}\right] \rightarrow \left[\begin{array}{ll} cat & X \\ f & \mathbf{a} \end{array}\right]$, dann terminiert die Berechnung der Chart nicht:

Die Regel S $\rightarrow$ X führt zu Generierung der Kante

$$k_{1'} = [0, 0, \left[\begin{array}{ll} cat & S \end{array}\right], e, \left[\begin{array}{ll} cat & X \\ f & a \end{array}\right], \{\ }].$$

Im nächsten Schritt generiert EXPAND$_{DAG}$ die Kante

$$k_{2'} = [0, 0, \left[\begin{array}{ll} cat & X \\ f & \boxed{12} \end{array}\right], e, \left[\begin{array}{lll} cat & X & \\ f & \left[\begin{array}{ll} f & \boxed{12} \end{array}\right] & \end{array}\right] \left[\begin{array}{ll} cat & Y \end{array}\right], \left\{\left[\begin{array}{ll} \boxed{12} & a \end{array}\right]\right\}],$$

die auch in die Chart eingetragen wird. Diese Kante aktiviert erneut EXPAND$_{DAG}$:

$$k_{3'} = [0,\ 0,\ \begin{bmatrix} cat & X \\ f & \boxed{13} \end{bmatrix},\ e,\ \begin{bmatrix} cat & X \\ f & \begin{bmatrix} f & \boxed{13} \end{bmatrix} \end{bmatrix} \begin{bmatrix} cat & Y \end{bmatrix},\ \left\{ \begin{bmatrix} \boxed{13} & \begin{bmatrix} f & a \end{bmatrix} \end{bmatrix} \right\}].$$

Da a $\not\preceq$ (f a) gilt auch: $k_{2'} \not\preceq k_{3'}$; d.h. k_3 wird in die Chart eingetragen und der Berechnungsprozeß *ad infinitum* fortgesetzt.

Um die Terminierung des Berechnungsprozeßes in jedem Fall zu garantieren, wurde in [**?**] vorgeschlagen, in Merkmalsstrukturen endliche Teilbereiche auszuzeichnen und nur diese Teilbereiche bei der Berechnung der Chart zu berücksichtigen. Endliche Teilbereiche werden durch die Formulierung von *Restriktoren* festgelegt. Ein Restriktor Φ kann als eine endliche Menge von Pfaden aufgefaßt werden.

Definition (13-2) *Beschränkte Merkmalsstruktur*

Die Beschränkung einer Merkmalsstruktur M auf einen Restriktor Φ (M$\uparrow \Phi$) ist die Merkmalsstruktur M' für die gilt:

1. M' $\preceq$ M;
2. für jeden Pfad p gilt entweder
   (a) M'(p) ist undefiniert;
   (b) M'(p) ist atomar oder
   (c) jedes Merkmal m $\in$ DOM(M'(p))[7] ist durch Φ legitimiert
   (d) es gibt keine andere Merkmalsstruktur, die Bedingung (1)-(2) erfüllt und von M' subsumiert wird.

Beispiel (13-9)

Wenn Φ durch die Pfadmenge {<a b>, <d e f>, <d i j f>} festgelegt ist (vgl. [**?**, S.147f.]) und

$$D = \begin{bmatrix} a & \begin{bmatrix} b & c \end{bmatrix} \\ d & \begin{bmatrix} e & \boxed{1} \begin{bmatrix} f & \begin{bmatrix} g & h \end{bmatrix} \end{bmatrix} \\ i & \begin{bmatrix} j & \boxed{1} \end{bmatrix} \\ k & l \end{bmatrix} \end{bmatrix}$$

dann ist

$$D \uparrow \Phi = \begin{bmatrix} a & \begin{bmatrix} b & c \end{bmatrix} \\ d & \begin{bmatrix} e & \boxed{1} \begin{bmatrix} f & [\,] \end{bmatrix} \\ i & \begin{bmatrix} j & \boxed{1} \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

Wie wir in den vorangegangenen Beispielen gesehen haben, bildet die Generierung der zyklischen Kanten den problematischen Fall; d.h., die Verwendung von Restriktoren ist nur innerhalb der Prozedur $\text{EXPAND}_{\text{DAG}}$ erforderlich:

[7] DOM(M'(p)) bezeichnet die Menge aller Attribute, die im Pfad p vorkommen.

PROZEDUR EXPAND$_{DAG}$'

DATEN: Eine kontextfreie Syntax G = <V_N, V_T, S, R> und eine Chart C.
EINGABE: Eine aktive Kante k = [i, j, A, α, Bβ, θ].

METHODE:
Für alle B' $\rightarrow \tau \in$ R:
Wenn < B $\sqcup$ B'>
Dann <*CLOSURE*([j, j, B' $\sqcup$ B$\uparrow$ Φ, e, τ, θ'])>

Die Wahl eines geeigneten Restriktors hängt grundsätzlich von der verwendeten Syntax ab; d.h., es wird in diesem Fall grammatikspezifisches Wissen benötigt, um den Parser zu steuern.

13.3 Implementierung

Wir beschränken uns darauf, an dieser Stelle nur die Funktionen für den Recognizer anzugeben. Aus Platzgründen haben wir darauf verzichtet, eine vollständige Implementierung der Subsumtionsrelation und eines nicht-destruktiven Algorithmus zur DAG-Unifikation anzugeben. Diese Programme können wie die übrigen in dem Buch abgedruckten Programme über uns bezogen werden (s. Vorwort).

Selektoren und Prädikate

Die Selektoren START, ENDE, KOPF, GESCHLOSSEN und OFFEN können aus Kapitel 7.5.2. übernommen werden. Wegen der geänderten Kanten- und Regelstruktur benötigen wir folgende zusätzliche Selektoren:

```
(defun variablen (kante)
  (fifth kante))
(defun linke-seite (regel)
  (caar regel))
(defmacro rechte-seite (regel)
  (cdar regel))

;;; AKTIVE-KANTE-P
;;; Das Prädikat testet, ob KANTE eine aktive Kante ist.
(defun aktive-kante-p (kante)
  (offen kante))
```