Syntaxgesteuerte Erkennung von gesprochenen Sätzen mit Methoden der Künstlichen Intelligenz

Diplomarbeit
ausgeführt am

Lehrstuhl für Datenverarbeitung
Technische Universität München
Prof. Dr.techn. J. Swoboda

von

Florian Schiel

Betreuer: Dr.-Ing. habil. G. Ruske

Beginn: Juni 1989
Abgabe: Dezember 1989

Zusammenfassung

In der automatischen Erkennung von fließend gesprochener Sprache ist es wegen der typischen Verschleifungen der Sprachmuster an den Wortgrenzen nicht möglich, die Lage der Wörter im Sprachsignal festzustellen. Bei Verwendung von Silben oder Halbsilben als kleinste Erkennungseinheiten läßt sich Anzahl und Lage der Silben eines gesprochenen Satzes mit einer gewissen Fehlerrate identifizieren. Man erhält also von Satzanfang bis Satzende eine Kette von Silben. Die Silben werden in Form von Lautsymbolen dargestellt und bilden das Erkennungsergebnis der ersten Stufe des Gesamtsystems. Je nach Größe des Wortschatzes ergibt sich für den gesprochenen Satz eine sehr hohe Anzahl von möglichen Wortkombinationen. Einzige Orientierungshilfe bei der Suche nach der tatsächlich gesprochenen Wortkombination ist die Ähnlichkeit der Lautsymbole von Lexikon-Wörtern mit Teilen der erkannten Silbenfolge. Diese akustische Ähnlichkeit läßt sich in Form von Kosten dergestalt ausdrücken, daß das ähnlichste lexikale Wort zu einem gesprochenen Wort auch die geringsten Kosten aufweist. Folglich hat auch die lexikale Wortkombination mit der geringsten Kostensumme die größte Ähnlichkeit mit dem gesprochenen Satz. Um die Anzahl der zu untersuchenden Wortkombinationen möglichst gering zu halten, wird ein Baumsuchverfahren entworfen, welches nur syntaktisch korrekte Sätze findet. Die Syntax-Kontrolle erfolgt dabei durch ein formales System in Form einer kontextsensitiven Phrasen-Struktur-Grammatik, welche einen Teilbereich der deutschen Grammatik abdeckt. Für dieses formale System wird in der Programmiersprache PROLOG ein Ableitungskalkül erstellt, das die Entscheidung 'syntaktisch richtig' oder 'syntaktisch falsch' für einen gegebenen deutschen Satz liefert. Um eine Realisierung des aufwendigen Baumsuchverfahrens zu ermöglichen, wird eine Abwandlung des A^*-Algorithmus als Suchstrategie verwendet, welche die auf die jeweilige Silbenzahl normierten Kosten zur Orientierung und optimistische Schätzkosten zur Beschneidung des Suchbaums verwendet. Die Schätzwerte entspringen dabei einer optimistischen Schätzung über den restlichen, noch nicht erkannten Teil des betrachteten Satzes. Das Suchverfahren ist zulässig in dem Sinne, daß garantiert die Wortkombination mit den geringsten Gesamtkosten gefunden wird, obwohl nicht alle möglichen Wortkombinationen überprüft werden müssen. Durch eine gezielte Korrektur der Schätzwerte wird das Suchverfahren noch einmal um den Faktor 13 beschleunigt. Das System weist eine deutliche Verbesserung der Satzerkennungsrate von 38 % (ohne Syntax) auf 74 % (mit Syntax) auf, es erfordert aber auch einen erheblich größeren Rechenaufwand.

Vorwort

"Den Stoff sieht jedermann vor sich;
den Gehalt findet nur der, der etwas
dazu zu tun hat, und die Form ist ein
Geheimnis den meisten."
J.W. Goethe, Maximen und Reflexionen

Automatische Erkennung fließend gesprochener Sprache bedeutet: eine Maschine soll eine hochkomplizierte Leistung erbringen, die jeder Mensch scheinbar mühelos beherrscht. Ist das überhaupt möglich ? Sich auf diese Frage einzulassen, bedeutet eine weitere Variante des ewigen Konflikts zwischen Materialismus und Idealismus, zwischen Holismus und Reduktionismus,... - die Liste läßt sich beliebig fortsetzen.
Deshalb vermeiden wir hier diese Frage und sagen:
Eine solche Leistung mit starken Einschränkungen muß möglich und für viele Anwendungen nützlich sein. Da wir hier vorerst diese Einschränkungen nicht weiter spezifizieren, wird uns darin kaum jemand widersprechen. Vieleicht erhellen sich diese Einschränkungen mit fortschreitender Erkenntnis über die Prinzipien der Spracherkennung von selbst.
Am Lehrstuhl für Datenverarbeitung der Technischen Universität München wird seit geraumer Zeit über dieses Thema geforscht. Die Erkennung von Vokalen, Konsonantenfolgen und, darauf aufbauend, von ganzen Wörtern ist so weit gediehen, daß nun auch die Erkennung von ganzen Sätzen begonnen wurde. Auf der Satzebene der Erkennung empfiehlt es sich, das allgemein vorhandene Wissen über die Syntax der deutschen Sprache zur Verbesserung der Erkennungsleistung so zu nutzen, daß nur syntaktisch richtige Sätze erkannt werden können. Auf diesem Gebiet wurde in dieser Gruppe bereits einiges erarbeitet und die vorliegenden Diplomarbeit stellt, so hoffe ich, einen weiteren Schritt in dieser Entwicklung dar.
Ich danke an dieser Stelle besonders dem Leiter der Forschungsgruppe Automatische Spracherkennung Dr.-Ing. habil. G. Ruske für die fachlich ausgezeichnete Betreuung meiner Arbeit. Darüber hinaus konnte ich weitgehend in eigener Regie arbeiten, was ich außerordentlich schätze. Darüber hinaus danke ich allen Mitgliedern des Lehrstuhls für das angenehme Arbeitsklima und das kollegiale Verhalten.

Inhaltsübersicht

1. Einführung

Die Aufgabe der automatischen Spracherkennung besteht darin, den Informationsgehalt einer akustischen Äußerung seitens des Menschen zu analysieren und in Form von geschriebenem Text auszugeben. Dieser bildet dann, je nach Anwendungsfall, die Grundlage für eine weitergehende Verarbeitung, z.B. elektronische Textverarbeitung, Steuerung von Maschinen, etc.
Ein System zur automatischen Erkennung gesprochener Sprache hat im allgemeinen mehrere hierarchisch angeordnete Ebenen. Auf der untersten Ebene wird das physikalische Sprachsignal für die Erkennung vorverarbeitet. Dabei werden meistens Merkmale mit hohem Informationsgehalt für die spätere Klassifikation gezielt extrahiert. Auf den folgenden Ebenen werden die Satz- und Silbengrenzen festgestellt und Lauteinheiten klassifiziert.

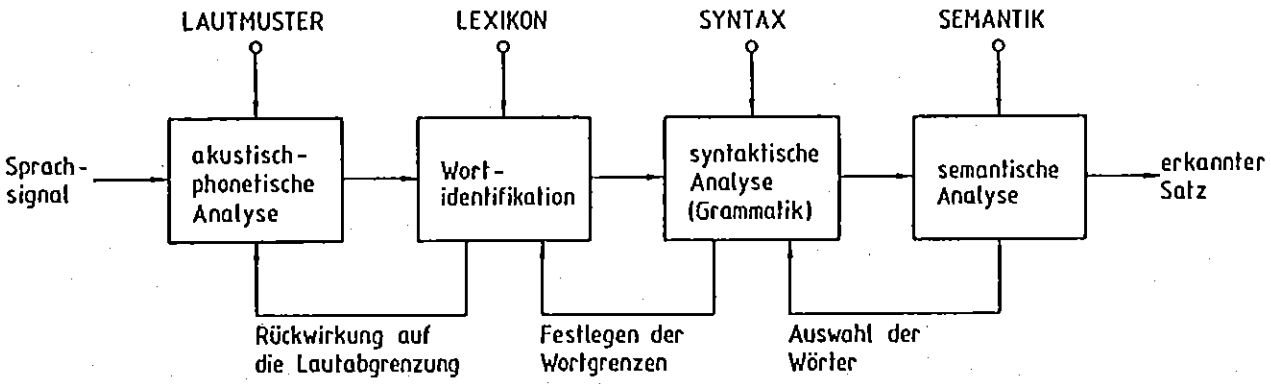


Bild 1.1. Struktur eines Spracherkennungssystems aus [Rus88], S.2

Für die Erkennung der einzelnen Wörter aus den Lauteinheiten stellt sich das Problem der nicht feststellbaren Wortgrenzen in fließend gesprochener Sprache. Daher werden die Wörter nicht einzeln und nacheinander klassifiziert, sondern der Kontext des gesamten Satzes berücksichtigt. Hier arbeiten zwei Ebenen, die der Worterkennung und die der Satzerkennung, Hand in Hand. Gesucht wird also nicht ein Wort nach dem anderen, sondern die beste Kombination von Wörtern, welche dann den erkannten Satz darstellt. Fehler, die auf niedrigeren Ebenen wie der Klassifikation der Lauteinheiten immer wieder auftreten, können auf diese Weise korrigiert werden. Zum einen dadurch, daß nur bestimmte Wörter überhaupt erkannt werden können und sinnlose Lautkombinationen wie "arfaq" im Repertoire des System nicht vorkommen,

zum anderen dadurch, daß syntaktisch falsche Sätze wie "hat bleiben mensch anfang französischen von" gar nicht erst in den Entscheidungsprozeß aufgenommen werden. Letzterer Punkt ist Kernthema dieser Arbeit: die Verbesserung der Spracherkennung durch Ausnutzung syntaktischen Wissens. Hat man damit die letzte Ebene erreicht ? Sicher nicht, aufbauend auf dem vorgestellten System sind Ebenen der semantischen und pragmatischen Prüfung denkbar. Dazu wäre allerdings ein sehr viel weitergehender Ansatz nötig, denn um ein System über Sinn und Kontextzugehörigkeit entscheiden zu lassen, ist es zunächst einmal notwendig, ein passendes maschinelles Welt- und Situationsbild zu schaffen. Ansätze innerhalb der Künstlichen Intelligenz (KI) zeigen in diesem Bereich bereits vielversprechende Ergebnisse ([Hof87], S. 624, [Win72], [Gro79]).

Die gesamte Vorverarbeitung und die unteren Ebenen des Systems bis hin zur Erkennung von einzelnen Wörtern sind nicht Inhalt dieser Arbeit. Im Literaturverzeichnis befinden sich Quellenangaben zu diesen Themen, insbesondere [Rus88] und [Wei].

In den Abschnitten 3 und 4 dieser Arbeit wird der Einsatz von Suchverfahren der Künstlichen Intelligenz für die Erkennung fließend gesprochener Sprache diskutiert. Es werden zwei Systeme SUCHE und SYNTAX definiert und begründet. Ein stichpunktartige Zusammenfassung dieser Systeme findet sich im Abschnitt 5. Abschnitt 6 beschäftigt sich mit den speziellen Problemen der Implementierung von beiden System auf einer MikroVAX II und enthält eine kurze Bedienungsanleitung des gesamten Systems.

In Abschnitt 7 schließlich werden die experimentellen Ergebnisse, die mit Hilfe der vorgestellten Suchverfahren erzielt wurden, vorgestellt und diskutiert.

2. Aufgabenstellung der Diplomarbeit

Die Ausgangsbasis für die Satzerkennung bildet ein silbenorientiertes Erkennungssystem ([Rus88],[Wei]).
Über mehrere Stufen der Verarbeitung erhält man ein symbolisches Abbild des gesprochenen Satzes in Form einer Computer-Lautschrift (ähnlich der internationalen Lautschrift), z.B. den Satz "Er tat es in dessen Namen":

/ @: R T A: T / @ S / I: N D @ S S 9 N N A: M M @ N

Die Lautfolge besteht aus einer Verkettung von Anfangskonsonantenfolge, Vokal und Endkonsonantenfolge (AVE), z.B. '/ @: R' = 'er' oder 'T A: T' = 'tat'.
Dadurch ist die Anzahl der erkannten Silben des Satzes bekannt, hier z.B. sind es acht Silben. Wegen der für fließend gesprochene Sprache typischen Verschleifung der Wortgrenzen sind diese aus dem Klassifikationsergebnis nicht zu erkennen, und es ist anschaulich klar, daß es selbst mit einem kleinen Vorrat an Wörtern sehr viele Wortkombinationen mit acht Silben gibt. Um zu prüfen, wie gut ein bestimmtes Wort des Lexikons an eine bestimmte Stelle in der Lautfolge paßt, steht ein Programm-Modul (WORSCORE) zur Verfügung, welcher mit Hilfe von dynamischer Programmierung einen Kostenwert für dieses Wort berechnet. Diese Kosten sind um so größer, je schlechter das Wort an der bezeichneten Position paßt, und sehr klein im umgekehrten Falle. Der Maximalwert (worst case) beträgt 1000 pro Silbe, der Minimalwert (best case) dagegen 1. Näheres zur Kostenberechnung findet sich im Abschnitt 3.3.
Das Problem besteht nun darin, möglichst schnell und effektiv die Wortkombination mit korrekter Silbenlänge zu finden, welche dem gesprochenen Satz entspricht. Dabei kann davon ausgegangen werden, daß der gesprochene Satz syntaktisch korrekt war. Es können daher alle Wortkombinationen, die syntaktisch falsche Sätze ergeben, aus dem Entscheidungsverfahren ausgeschlossen werden. Desweiteren darf angenommen werden, daß die unteren Ebenen des Spracherkennungssystems immerhin so zuverlässig arbeiten, daß ein korrektes Wort an der richtigen Stelle mit hoher Wahrscheinlichkeit auch die niedrigsten Kosten ergibt. Dann läßt sich das Problem umformulieren in: Gesucht ist die Wortkombination mit korrekter Silbenzahl, die einerseits einen syntaktisch richtigen

Satz ergibt, und andererseits von allen möglichen, syntaktisch richtigen Sätzen die kleinste Summe der einzelnen Wortkosten aufweist. Was hier angestrebt wird, ist also keine sequentielle Erkennung Wort für Wort, sondern die Erkennung unter Berücksichtigung des Satzkontexts. Man könnte von auch von kontextsensitiver Erkennung sprechen.

Die Aufgabenstellung der vorliegenden Arbeit ist die Erstellung eines geeigneten Suchverfahrens, welches die oben genannten Aufgaben möglichst effektiv und sicher löst. Insbesondere sollen in der Literatur bekannte Baumsuchverfahren für diese Anwendung getestet und eventuell eigene Konzepte entwickelt werden. Die verschiedenen Suchverfahren sollen auf einer MikroVAX II implementiert und anhand des am Lehrstuhl für Datenverarbeitung existierenden Sprachmaterials getestet werden.

3. Problemlösung mit Methoden der Künstlichen Intelligenz (KI)

3.1. Allgemeines über KI-Systeme

Es würde zu weit führen, hier einen umfassenden Abriß der gängigen KI-Methoden zu geben. Da jedoch das vorliegende Problem eines der Zentralthemen der KI darstellt und im weiteren Verlauf häufiger Gebrauch von Strukturen und Begriffen der KI gemacht wird, soll hier kurz die gängigste Struktur eines solchen Systems vorgestellt werden ([Nil82], S. 17 ff).

In den meisten KI-Systemen lassen sich folgende Elemente identifizieren: Datenbasis, Produktionsregeln, Strategiekontrolle und Terminal-Bedingung. Die **Datenbasis** stellt gewissermaßen das aktuell erarbeitete Wissen des Systems dar. Es wird in strukturierter Form gespeichert, um für die spätere Erarbeitung von neuem Wissen als Grundlage zu dienen. Bei den meisten Systemen ist beim Start die Datenbasis leer, da die Daten spezifisch für eine individuelle Aufgabe sind und beim Stellen einer neuen Aufgabe natürlich gelöscht werden müssen. Die **Produktionsregeln** haben die Aufgabe, aus Teilen der Datenbasis neue Elemente der Datenbasis zu erarbeiten. In ihnen ist das globale Wissen über das Problemgebiet enthalten, welches auf jedes individuelle Problem anwendbar ist. Sie kombinieren quasi vorhandenes Wissen aus der Datenbasis mit ihrem inhärenten globalen Wissen und erzeugen so eine Bereicherung der Datenbasis. Die **Strategiekontrolle** entscheidet, welche Produktionsregeln auf welche Teile der Datenbasis angewandt werden sollen. Damit ein System effektiv arbeitet, sollte auch die Strategiekontrolle globales Wissen über das Problemgebiet enthalten, damit möglichst nur solches Wissen in die Datenbasis gelangt, welches für die Problemlösung relevant ist. Es gibt grundsätzlich zwei verschiedene Typen der Strategiekontrolle: Einbahnstraßen-Systeme und tastende Systeme. Ein Einbahnstraßen-System kann einen einmal eingeschlagenen Pfad auf der Suche nach Lösungen nicht mehr verlassen. Dagegen können tastende Systeme, wenn sich der eingeschlagene Pfad als ungünstig erweist, diesen wieder zurückgehen (Backtracking) oder auch ganz verlassen (Graphen-Suche) und nach alternativen Wegen suchen. Die **Terminalbedingung** schließlich legt fest, wann ein Teil der Datenbasis das gewünsch-

Ergebnis enthält und die Suche erfolgreich abgebrochen werden kann. Eventuell entscheidet sie auch über einen erfolglosen Abbruch der Suche. Die Arbeitsweise eines KI-Systems gestaltet sich nun folgenderweise: Die Strategiekontrolle wählt eine Produktionsregel und einen Teil der Datenbasis aus, welche sie für das Auffinden der Lösung für geeignet hält. Die gewählte Produktionsregel erzeugt (oder auch nicht) neue Elemente in der Datenbasis. Die Terminalbedingung prüft diese neuen Elemente, ob sie eventuell die gewünschte Lösung enthalten. In diesem Falle bricht sie die Suche ab, im anderen Falle wählt die Strategiekontrolle eine neue Regel aus, usw.

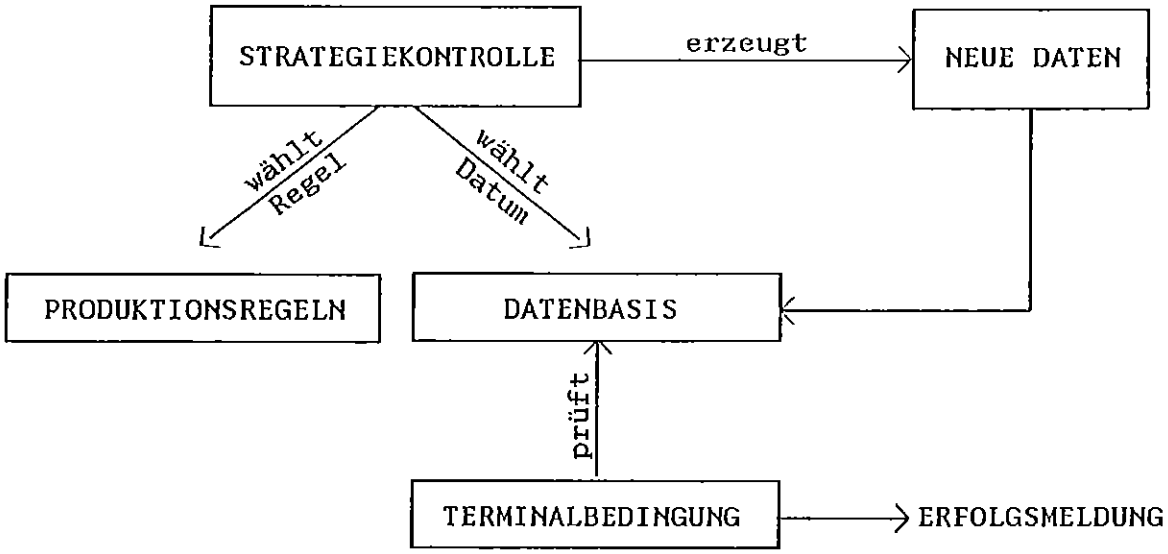


Bild 3.1. Allgemeine Struktur eines KI-Systems

Die hier vorgestellte Form ist noch sehr allgemein und läßt sich, wie gesagt, für fast alle KI-Systeme heranziehen. Im folgenden sollen für ein System namens SUCHE einige zusätzliche Einschränkungen gemacht werden: Es gibt nur eine einzige Produktionsregel, welche auf genau ein Element der Datenbasis angewandt werden kann, d.h. die Strategiekontrolle muß nur noch das Element der Datenbasis bestimmen, aber keine Regel dazu. Ein Element der Datenbasis kann nur genau einmal erzeugt und wieder gelöscht werden. Es soll eine tastende Strategiekontrolle, genauer eine Graphen-Suche verwendet werden.

3.2. Der Suchbaum von SUCHE

Wir haben bis jetzt die wichtigsten Elemente des System SUCHE, nämlich Datenbasis, Produktionsregeln, Strategiekontrolle und Terminalbedingung, grob definiert und - im Hinblick auf unsere Anwendung - spezifiziert.

Um nun die vorgestellten Elemente etwas plastischer erscheinen zu lassen, kann man sich den Suchprozeß als das Wachsen eines Baumes vorstellen. Der Baum besteht aus einer Wurzel, aus Ästen, Knoten und Blättern. Er ist somit der Spezialfall eines Graphen, in welchem jeder Knoten nur genau einen Vaterknoten hat. Die Wurzel ist die leere Datenbasis, Äste bezeichnen die Anwendung einer Produktionsregel, offene Knoten und Blätter sind Elemente der Datenbasis, wobei Blätter einfach Knoten sind, aus denen keine Äste sprießen können. Würde die Strategiekontrolle, ohne ihr globales Wissen zu nutzen, stur jeden Knoten expandieren, d.h. die Produktionsregel auf ihn anwenden, so erhielte man den expliziten Suchbaum des Problemgebiets. Da der Aufwand für eine solche Berechnung i.a. viel zu groß ist, wird die Strategiekontrolle nur solche Knoten expandieren, von denen sie weiß oder vermutet, daß sie zum Ziel führen, d.h. ein Zielblatt b erzeugen, welches die Terminalbedingung erfüllt. Das Wachsen des Baumes wird also nicht auf breiter Front erfolgen, sondern möglichst rasch innerhalb eines schmalen Streifens auf das Zielblatt zustreben. Die Strategiekontrolle benötigt natürlich zur Beurteilung von verschiedenen Wachstumslinien Informationen. Zu diesem Zweck werden die Äste mit Kosten belegt, d.h. die Anwendung der Produktionsregel und Erzeugung eines neuen Knotens kostet etwas. Je vielversprechender dieser neue Knoten ist, desto niedriger seien die Kosten zu seiner Erzeugung. Da zu jedem Knoten n von der Wurzel ein eindeutiger Pfad führt, läßt sich für jeden Knoten die Summe der Kosten aller Äste auf diesem Pfad angeben. Dieser Wert heiße Gesamtkosten g(n) des Knotens n. Das Zielblatt b hat die niedrigsten Gesamtkosten von allen möglichen Blättern t_i:

$$g(b) < g(t_i) \qquad (3.2.1)$$

Die Aufgabe des Systems läßt sich also umformulieren zu:
Finde das Blatt im Suchbaum, welches den billigsten Pfad von der Wurzel

aufweist.

Man beachte, daß der explizite Suchbaum für jedes individuelle Problem zwar die gleiche Form, aber unterschiedliche Kosten an den Ästen hat. Da diese Kosten letztlich entscheidend für das Finden des Zielblattes b sind, genügt es nicht, einmal in einem gewaltigen Rechenaufwand die Form des Suchbaums für ein Problemgebiet zu ermitteln, sondern vielmehr muß für jedes individuelle Problem schrittweise ein Partialbaum inklusive Kosten erstellt werden, welcher das Zielblatt b enthält.

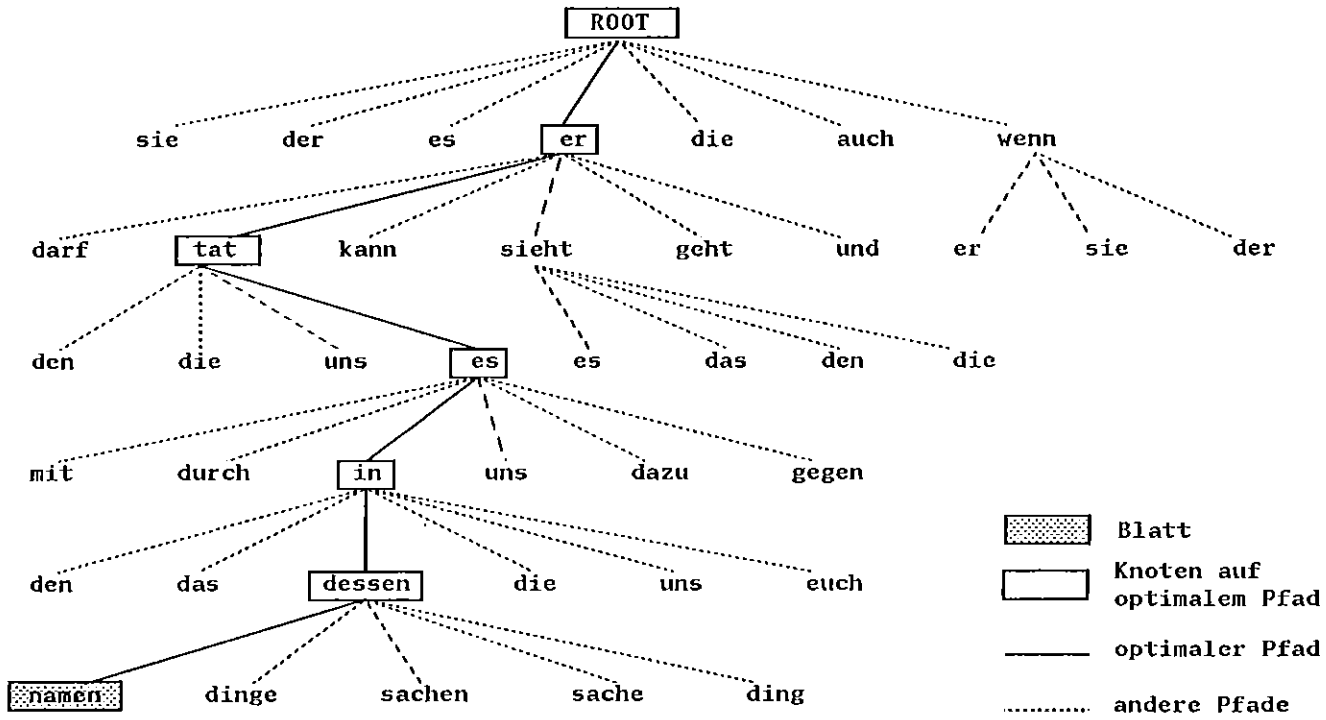


Bild 3.2 Partialbaum des Satzes "Er tat es in dessen Namen."

3.3. Die Kosten eines Satzes

Im vorhergehenden Abschnitt wurde der Kostenbegriff als Entscheidungskriterium eingeführt. Da im folgenden verschiedene Kostenbegriffe verwendet werden, soll in diesem Abschnitt die Berechnung dieser Kosten geklärt werden.

Basis für alle Kostenberechnungen sind die Kosten des einzelnen Wortes.

Diese sollen ausdrücken, wie gut ein bestimmtes Wort des Lexikons auf eine bestimmte Silbenposition der Lautfolge paßt. Die Aussprachevarianten aller Lexikonwörter sind in einem sog. Wortmodell-Lexikon in symbolischer Form gespeichert. Aus einer Trainingsphase sind die Rückschlußwahrscheinlichkeiten für das Auftreten von Silbeneinheiten (Anfangskonsonantenfolge, Vokal, Endkonsonantenfolge) $p(x|y)$ bekannt. $p(x|y)$ drückt die Wahrscheinlichkeit aus, daß die Einheit x gesprochen wurde, wenn die Einheit y in der Lautfolge auftritt. Da die Kosten eines ganz bestimmten Wortes aus dem Lexikon berechnet werden sollen, ist x an jeder Stelle des Wortes aus dem Wortmodell-Lexikon bekannt. y erhalten wir aus der entsprechenden Position in der Lautfolge. Wir erhalten also pro Silbe drei verschiedene Rückschlußwahrscheinlichkeiten für Anfangskonsonantenfolge, Vokal und Endkonsonatenfolge. Hat das Lexikonwort s Silben, so ergibt das 3s Werte. Diese werden im Falle, daß das Lexikonwort wirklich an der bestimmten Silbenposition gesprochen wurde, relativ hoch, im umgekehrten Falle niedrig sein. Geht man in erster Näherung davon aus, daß das Auftreten der einzelnen Einheiten statistisch unabhängig ist, erhält man durch Multiplikation der Rückschlußwahrscheinlichkeiten aller Einheiten einer Silbe die Wahrscheinlichkeit für das Auftreten einer Silbe S an einer bestimmten Silbenposition:

$$p(S) = p(x|y)_A \quad p(x|y)_V \quad p(x|y)_E \tag{3.3.1}$$

Die Wahrscheinlichkeit $p(S)$ kann Werte in Promille zwischen 10 und 1000 annehmen und läßt sich daher durch Inversion direkt in Kosten pro Silbe $k(S)$ umrechnen

$$k(S) = 1500 - 500 \log(p(S)), \tag{3.3.2}$$

welche genau den Bereich von 0 bis 1000 überstreichen. Der Fall, daß $k(S) = 0$ ist, darf aus Gründen, die später noch klar werden, nicht auftreten, daher wird in diesem Falle $k(S) = 1$ gesetzt. Da es sich bei $k(S)$ auch nach der Umrechnung in Kosten um eine logarithmische Größe handelt, erhält man die lokalen Kosten $k(W)$ für ein bestimmtes Wort W an einer bestimmten Silbenposition einfach durch Addition der einzelnen Silbenkosten $k(S_i)$

$$k(W) = \sum_i k(S_i) \qquad (3.3.3)$$

Eine detailiertere Beschreibung entnehme man sinngemäß [Rus88], S. 155 ff. Um nun zu den Kosten eines Satzes oder Teilsatzes zu kommen, werden die Kosten der einzelnen Wörter $k(W_j)$ aufsummiert

$$g(Satz) = \sum_j k(W_j) \qquad (3.3.4)$$

Diese, mit der Anzahl j der Wörter streng monoton steigende Funktion sei im folgenden Gesamtkosten des Satzes oder Teilsatzes genannt. Um auch Sätze verschiedener Silbenzahl vergleichen zu können, werden die Gesamtkosten auf die Anzahl der Silben im Satz z normiert:

$$n(Satz) = g(Satz)/z(Satz) \qquad (3.3.5)$$

Die Funktion n(Satz) wird normierte Kosten des Satzes oder Teilsatzes genannt.
Angenommen eine Lautfolge enthält 6 Silben und der Teilsatz 'Der Mensch denkt...' wird gerade expandiert. Die Gesamtkosten g(Teilsatz) und normierten Kosten n(Teilsatz) sind berechnet und können als Grundlage für Entscheidungen der Strategiekontrolle dienen. Interessant wäre jetzt zu erfahren, wieviel der Teilsatz in Zukunft mindestens noch kosten wird. Auf jeden Fall können die Gesamtkosten des kompletten Satzes g(Satz) nicht kleiner sein als die Gesamtkosten g(Teilsatz) des obigen Teilsatzes, da g eine streng monoton steigende Funktion ist. Darüber hinaus wissen wir, daß von den 6 Silben des Satzes noch 3 unerkannt sind. Für diese 3 Silben läßt sich eine 'optimistische Restschätzung' h(Restsatz) angeben, welche besagt, wieviel diese restlichen drei Silben mindestens kosten müssen. Dabei geht man im Prinzip genauso vor, wie bei der Berechnung der Kosten eines Wortes k(W) (s.o.), aber mit folgenden Randbedingungen:

- der ganze Rest der Lautfolge wird als ein Wort W aufgefaßt.
- es findet kein Vergleich mit dem Wortmodell-Lexikon statt, sondern für jede Einheit y, die in der Lautfolge auftritt, wird die Rückschlußwahrscheinlichkeit p(y|y) in (3.3.1) verwendet. In Worten: die Wahr-

scheinlichkeit, daß bei Auftreten der Einheit y auch diese Einheit y gesprochen wurde. Der Optimismus liegt also darin, anzunehmen, daß die gesamte Vorverarbeitung und der Klassifikator keinen Fehler gemacht haben. Man erhält einen Kostenwert h(Restsatz) für den noch nicht erkannten Teil des Satzes, welcher selbst bei optimaler Erkennung niemals unterschritten werden kann. Wird h(Restsatz) zu g(Teilsatz) addiert, so erhalten wir die Schätzkosten f(Teilsatz), die auf jeden Fall kleiner als die Gesamtkosten g(Satz) des vollständigen Satzes sind:

$$f(\text{Teilsatz}) = g(\text{Teilsatz}) + h(\text{Restsatz}) < g(\text{Satz}) \qquad (3.3.6)$$

Damit sind alle wichtigen Kostenfunktionen für das System SUCHE definiert. Da der Suchbaum von Abschnitt 3.2. ein geeignetes Mittel darstellt, verschiedene Suchprozesse darzustellen, möchte ich in Zukunft die verschiedenen Kosten als Funktionen von Knoten ansehen. Ist n ein beliebiger Knoten des partiellen Suchbaums, so sei

- k(n) : lokale Kosten des durch diesen Knoten n angehängten Wortes.
- g(n) : Gesamtkosten aller Knoten auf dem eindeutigen Suchpfad von der Wurzel bis n, k(n) inklusive.
- n(n) : g(n) normiert auf die Anzahl der Silben im Teilsatz oder Satz, der durch n repräsentiert wird.
- h(n) : Optimistische Restschätzung über die restlichen Silben der Lautfolge nach dem Teilsatz, der durch n repräsentiert wird.
- f(n) : Optimistische Schätzkosten des gesamten Satzes, der aus dem Knoten n sprießen kann, f(n) = g(n) + h(n)

3.4. Definition der Datenbasis von SUCHE

Wir haben gesehen, daß die Datenbasis den aktuell erarbeiteten Wissensschatz über ein individuelles Problem darstellt und als die Menge aller noch nicht expandierten Knoten in einem partiellen Suchbaum interpretiert werden kann.

Die Form der Datenbasis und damit der Knoten für das System SUCHE werden wie folgt definiert:

Die Wurzel des Suchbaums soll den leeren Satz darstellen. Die Knoten der ersten Ebene seien Teilsätze mit genau einem Wort, die Knoten der zweiten Ebene seien Teilsätze mit genau zwei Wörtern, usw. Ein Knoten soll also vorerst folgende Daten enthalten:

- Die Wortkette, die von der Wurzel zu ihm führt, im weiteren als Teilsatz bezeichnet.
- Die aufsummierten Kosten aller zwischen Wurzel und Knoten liegenden Äste (Gesamtkosten g(n)).
- Die Anzahl der Silben des Teilsatzes.

Dies ist sozusagen die Minimalausstattung eines Knotens. Mit diesen Informationen läßt sich bereits eine erfolgreiche Suche durchführen. Z.B. könnte eine uninformierte Strategiekontrolle einfach alle Knoten expandieren, bis nur noch Blätter (Knoten mit korrekter Silbenzahl) im Baum existieren, und das Blatt b mit dem geringsten g(b) als Lösung ausgeben. Um wirklich effektive Suchprozesse zu erzeugen, wird diese Grundausstattung noch erweitert durch:

- Gesamtkosten normiert auf Silbenzahl n(n).
- Schätzkosten f(n) = g(n) + h(n).

3.5. Erste Definition der Produktionsregel von SUCHE

Aufgabe der Produktionsregel ist es, aus einem Teil der Datenbasis neues Wissen für die Datenbasis zu erzeugen, oder, in Baum-Terminologie gesprochen, aus einem Knoten Äste zu neuen Knoten oder Blättern wachsen zu lassen.

In Anwendung auf unser Erkennungsproblem bedeutet dies folgendes: Ein Knoten, der von der Strategiekontrolle zur Expansion ausgewählt wurde, enthält einen bestimmten Teilsatz, dessen Silbenzahl und diverse Kosten. Seine Nachfolgerknoten enthalten genau diesen Teilsatz um ein Wort erweitert. Die Kosten dieser Erweiterung müssen berechnet und zu den Gesamtkosten des Vaterknotens addiert werden. Desgleichen muß die Silbenzahl des neuen Wortes ermittelt und zur Silbenzahl des Vaterknotens

addiert werden. Mit Hilfe des Moduls WORDSCORE (s. Abschnitt 2.) ist beides kein Problem.
Die Frage ist nur: welche Wörter sollen denn überhaupt als Erweiterung des Teilsatzes in Frage kommen ? Eine sehr einfache Produktionsregel könnte sagen: alle Wörter des Lexikons. Damit würde aber selbst ein Partialbaum bald alle Speichergrenzen sprengen und die Zahl der expandierbaren Knoten explodieren. Eine intelligenteres System dagegen läßt nur Erweiterungen zu, die zu syntaktisch korrekten Teilsätzen führen, und schränkt somit die Zahl der Nachfolgerknoten erheblich ein. Fassen wir also zusammen, welche Aufgaben unsere Produktionsregel erfüllen muß:

- Auswahl der syntaktisch richtigen Satzerweiterungen aus Lexikon.
- Berechnung der Kosten, Gesamtkosten und neuen Gesamtsilbenzahl der neuen Knoten.

Man beachte, daß die Anzahl der Nachfolgeknoten durch den Teilsatz des Vaterknotens und die Syntaxkontrolle der Produktionsregel bestimmt wird. Hat ein Teilsatz laut Syntaxkontrolle keinen möglichen Nachfolger im Lexikon, so wird der expandierte Knoten aus der Datenbasis gestrichen, ohne Nachfolger zu erzeugen. Das Problem des Syntax-Tests von deutschen Teilsätzen möchte ich hier noch offen lassen, um später in Abschnitt 4. detailiert darauf zurückzukommen. Vorerst wollen wir annehmen, daß unsere Produktionsregel alle obigen Aufgaben lösen kann.

3.6. Definition der Terminalbedingung von SUCHE

Die Terminalbedingung muß für ein Element der Datenbasis erfüllt sein, damit der Suchprozeß erfolgreich abbrechen kann. Sie muß während der Suche für alle Erweiterungen der Datenbasis geprüft werden.

Bevor wir zur Strategiekontrolle kommen, die einen breiteren Raum in dieser Arbeit einnehmen wird, soll die Terminalbedingung für unsere Anwendung festlegt werden. Ein betrachtetes Blatt des partiellen Suchbaums erfüllt die Terminalbedingung genau dann, wenn

- der in ihm enthaltene Teilsatz in der Silbenzahl mit der Lautfolge übereinstimmt. (Diese Bedingung ist eigentlich

trivial, da es sich sonst gar nicht um ein Blatt handeln würde. Der Suchprozeß gestaltet sich jedoch einfacher, wenn zwischen Knoten und Blättern nicht unterschieden wird).
- der in ihm enthaltene Teilsatz darüber hinaus ein syntaktisch richtiger, abgeschlossener Satz ist.
- sichergestellt werden kann, daß sich im impliziten Suchbaum des Erkennungsproblems kein Blatt finden läßt, dessen Gesamtkosten geringer sind als die des betrachteten Blattes.

Einige Bemerkungen dazu: Es ist durchaus möglich, Wortketten zu bilden, die einerseits einen syntaktisch richtigen Teilsatz und zugleich einen syntaktisch richtigen, abgeschlossenen Satz bilden. Z.B. ist die Wortkette 'Der Mensch denkt' ein abgeschlossener Satz und zugleich ein Teilsatz für den Satz 'Der Mensch denkt nach'. Da wir jedoch nur wissen, daß die Wortkette in b als Teilsatz syntaktisch korrekt ist (sonst hätte die Produktionsregel ihn nicht erzeugt), muß die Terminalbedingung zusätzlich prüfen, ob es sich auch um einen syntaktisch korrekten, abgeschlossenen Satz handelt.
Die dritte Bedingung ist sicher am schwersten zu prüfen. Eine schlecht informierte Strategiekontrolle muß zuerst alle möglichen Blätter im Suchbaum finden, bevor die Terminalbedingung überhaupt geprüft werden kann.

3.7. Suchstrategien

3.7.1. Allgemeines

In Anschnitt 3.1. wurden bereits die verschiedenen Typen der Strategiekontrolle angesprochen. Man unterscheidet zwischen Einbahnstraßen-Strategien (irrevocable strategy) und tastenden Strategien (tentative strategy, trial-and-error-search). Erstere umfassen Suchstrategien wie Gradientenverfahren, hill-climbing etc. Da solche Strategien einen einmal gemachten Fehler nicht mehr korrigieren können, sind sie für unserere Anwendung grundsätzlich weniger geeignet, da im Suchbaum wegen fehlerhafter Worterkennung oft zunächst der falsche Suchpfad eingeschlagen wird. Tastende Strategien lassen sich grob in zwei weitere Klassen einteilen: Kann eine Suchstrategie im Suchbaum immer nur zu Sohn- oder Vaterknoten wechseln, so spricht man von Backtracking, kann sie dagegen einen Suchpfad auch völlig verlassen und in einen beliebigen, anderen Knoten wieder einsteigen, so handelt es sich um eine Graphensuche. Backtracking Strategien, wie z.B. der branch-and-bound ([Sha66]), haben den Vorteil, daß sie sehr effektiv arbeiten. In PROLOG z.B. ist auch ein automatischer Backtracking-Algorithmus für die Abarbeitung der Prädikate verantwortlich. Andererseits ist es schwieriger, globales Wissen in einem Backtracking-Verfahren erfolgreich einzusetzen, da die Suchstrategie nicht zwischen beliebigen Knoten springen kann. Nehmen wir an, eine tastende Suchstrategie befindet sich gerade im Knoten n des partiellen Suchbaums. Der Knoten wird expandiert in i Nachfolger n_i und es ergibt sich, daß deren Kostenfunktionen $b(n_i)$ erheblich schlechter sind als von n: $b(n_i) > b(n)$. b(n) sei hier eine beliebige, geeignete Kostenfunktion. In einem anderen Ast des partiellen Suchbaums befinde sich ein Knoten x, dessen Kostenfunktion niedriger ist als die aller Nachfolger n_i von n: $b(x) < b(n_i)$. Leider ist aber x nicht der Vater von n. Eine Backtracking-Strategie kann nun nicht erkennen, daß ein besserer Ansatz zur Weitersuche existiert, da sie quasi über den Horizont der sie unmittelbar umgebenden Knoten nicht hinaussieht. Sie kann allenfalls die Nachfolger n_i mit dem Vater v des Knotens n vergleichen, und falls $b(v) < b(n_i)$, nach v zurückkehren, anstatt den besten der n_i auszuwählen. Vieleicht ergeben sich von v aus bessere Möglichkeiten. Eine Graphen-Suche dagegen wäre ohne weiteres in der Lage, den Knoten n zu verlassen, direkt nach x zu springen und die

Suche von dort aus weiterzuführen. Die Knoten n_i bleiben dann als 'offene Knoten' in der Datenbasis, um eventuell später expandiert zu werden. Da der soeben geschilderte Fall in unserer Anwendung häufig auftritt, soll das System SUCHE eine Graphen-Suche, genauer gesagt eine Baum-Suche als Strategiekontrolle erhalten.

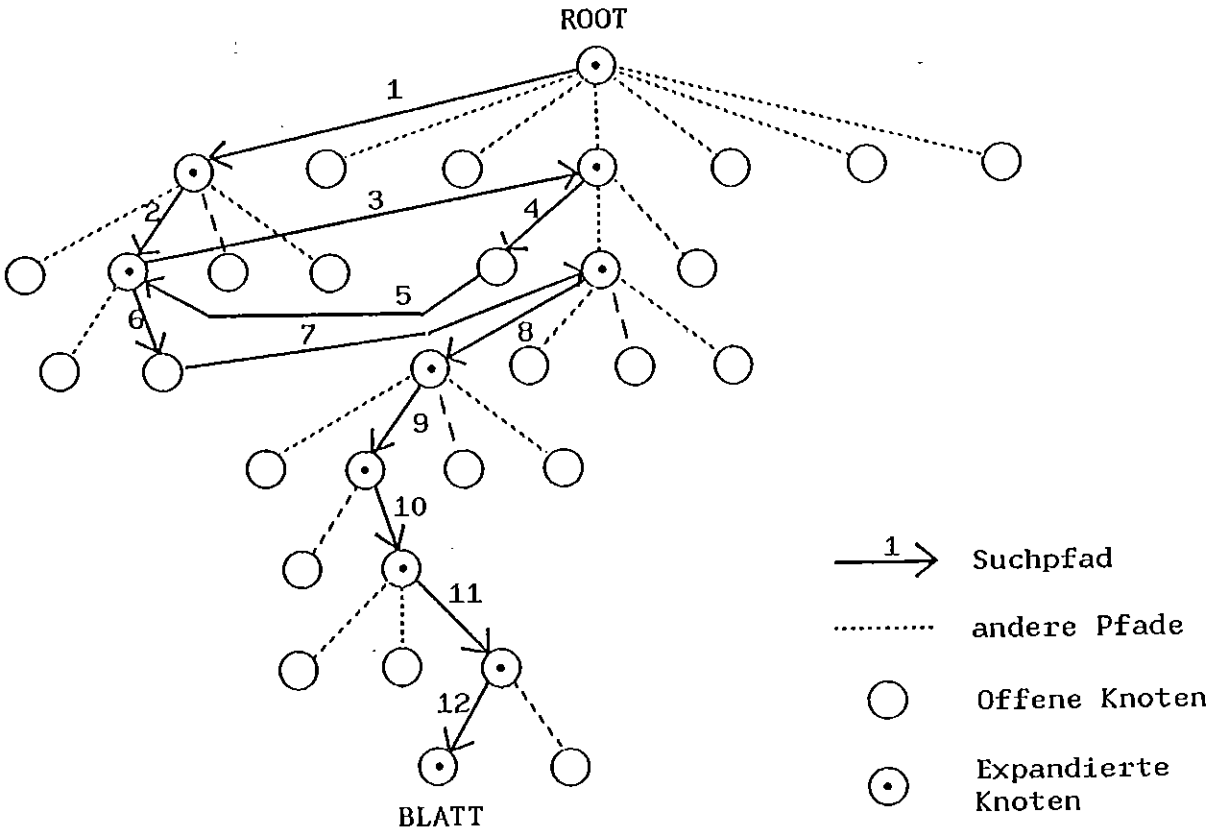


Bild 3.3 Suchpfad einer Graphensuche, Sprünge zwischen nicht-benachbarten Knoten sind möglich

Noch ein Wort zur Nomenklatur: Man spricht von einer zulässigen Suchstrategie, wenn gezeigt werden kann, daß das Suchverfahren immer im Blatt b mit den niedrigsten Gesamtkosten g(b) endet. Ein zulässiges Suchverfahren ist damit z.B. eine uninformierte Breitensuche, die einfach alle möglichen Blätter berechnet und das beste Blatt b heraussucht. Eine unzulässige oder nicht optimale Strategie dagegen garantiert nur, daß überhaupt ein Blatt als Lösung gefunden wird, jedoch nicht, daß dieses auch die beste Lösung ist.

3.7.2. Der A^*-Algorithmus

Eine wichtige, zulässige Suchstrategie zum Absuchen eines impliziten Suchbaums ist der A^*-Algorithmus. Da der A^* die Grundlage bildet für das später beschriebene Suchverfahren DFA^* (Abschnitt 3.7.3.), soll er hier kurz beschrieben werden. Eine detailiertere Beschreibung sowie der Beweis der Zulässigkeit des A^* finden sich z.B. in [Nil82], S. 72 ff.

Für die Anwendung des A^* muß für jeden expandierten Knoten bzw. jedes Blatt n des partiellen Suchbaums eine Gesamtkostenfunktion g(n) berechenbar sein, welche die Summe der Kosten aller Äste von der Wurzel zu n darstellt. Das gesuchte Zielblatt b hat die kleinsten Gesamtkosten g(b) von allen Blättern des impliziten Suchbaums. Außerdem existiert eine Schätzfunktion h(n), welche für die Kosten aller Restpfade aus n zu den Blättern b_i, welche aus n sprießen können, einen unteren Grenzwert darstellt. Mit anderen Worten: kein Pfad vom Knoten n zu seinen möglichen Nachfolgeblättern b_i kann billiger sein als h(n). Der A^* berechnet nun zu jedem Knoten n, den er expandiert, die Schätzkostenfunktion

$$f(n) = g(n) + h(n) \qquad (3.7.1)$$

Damit gilt automatisch

$$f(n) < g(b_i) \qquad (3.7.2)$$

Die Strategiekontrolle eines A^* läßt sich folgendermaßen formulieren:

"Expandiere den Wurzelknoten w. Dieser enthält kein Wort und alle seine Kosten sind Null. Die Nachfolge-Knoten bilden die Menge OFFEN.

Wiederhole bis Terminalbedingung eintritt:

Expandiere immer den Knoten n aus der Menge OFFEN als nächstes, dessen Schätzkostenfunktion f(n) ein Minimum von allen offenen Knoten ist.
Entferne dann den Knoten n aus der Menge OFFEN und füge alle seine Nachfolger, Knoten und Blätter, deren Schätzkosten kleiner sind als

die Gesamtkosten eines bereits gefundenen Blattes, der Menge OFFEN bei.
Wenn ein neues Blatt auftaucht, entferne alle Elemente aus OFFEN, deren Schätzkosten schlechter sind als die Gesamtkosten des neuen Blattes.

Gib das einzige Element aus OFFEN als Lösung aus."

Die Terminalbedingung des A^* wird zu:

"Die Menge OFFEN enthält nur noch ein Element und dieses ist nicht der Wurzelknoten w."

Dazu eine kurze, informale Begründung: Hat der A^* ein Blatt b gefunden, das nicht das gesuchte Zielblatt z ist, so gilt per definitionem

$$g(z) < g(b) \qquad (3.7.3)$$

Jeder Knoten n, dessen Schätzfunktion f(n) größer ist als die Gesamtkosten g(b) des gefundenen Blattes b, kann niemals zu einem Blatt b1 führen, das geringere Kosten aufweist als das bereits gefundene Blatt b, denn es gilt

$f(n) < g(b1)$ wegen (3.7.2),

$g(b) < f(n)$ (s.o.) und damit

$$g(b) < g(b1) \qquad (3.7.4)$$

Damit kann aber ein Pfad durch n wegen (3.7.4) niemals zu z führen und somit ist es sinnlos, den Knoten n jemals wieder zu expandieren, er kann also aus der Menge OFFEN entfernt werden.
Hat dagegen der A^* bereits das Zielblatt z gefunden, wird er alle Elemente mit schlechteren Schätzkosten aus der Menge OFFEN entfernen und fortfahren, die restlichen Knoten zu expandieren. Da diese aber alle auf Suchpfaden liegen, die nicht zu z führen (zu z führt ja nur ein eindeutiger Suchpfad !), werden die Schätzkosten ihrer Nachfolger über kurz

oder lang schlechter werden als g(z) und somit auch aus der Menge OFFEN verschwinden. Zurück bleibt allein das Zielblatt z.
Ist gewährleistet, daß g(n) streng monoton steigend ist, d.h.

$$g(n) < g(n_i) \qquad \text{mit: } n_i \text{ sind Nachfolger zu } n \qquad (3.7.5)$$

dann läßt sich zeigen, daß der A^* eine zulässige Strategie ist ([Nil82], S. 76). Die Mächtigkeit des A^* liegt natürlich in der heuristischen Funktion h(n). Je näher h(n) an den tatsächlichen Restwert bis zum besten Blatt h*(n) herankommt, desto erfolgreicher der A^*. Eine triviale Restschätzung wäre h(k) = 0 für alle offenen Knoten k. Der A^* orientiert sich in diesem Falle nur an den Gesamtkosten g(n) und wird so zur Breitensuche (breadth first). Denn ein solcher Suchprozeß expandiert i.a. sehr gleichmäßig Ebene für Ebene im Suchbaum und führt zu erheblichem Rechenaufwand. Die optimale Restschätzung wäre h(n) = h*(n). Ein solcher A^* geht niemals in die Irre, sondern findet auf Anhieb den optimalen Pfad von der Wurzel zum Zielblatt b.
Der A^* wurde für unsere Problemstellung implementiert (siehe Dokumentationsband, Abschnitt 1.3.). Dazu wurde die in Abschnitt 3.3. vorgestellte optimistische Restschätzung für h(n) eingesetzt. Leider zeigten Versuche, daß trotz A^* eine extreme Breitensuche einsetzt, welche den vorhandenen Speicherplatz von ca. 12 MByte rasch überschreitet (vgl. Abschnitt 7.2.2.).

3.7.3. Definition der Strategiekontrolle von SUCHE: DFA*

Im vorangegangenen Abschnitt wurden die Vor- und Nachteile des reinen A^* für unsere Anwendung kurz beleuchtet. In diesem Abschnitt wird schrittweise eine Strategiekontrolle Depth-First-A^* (DFA^*) für das System SUCHE definiert, welche die bislang besten Erkennungsergebnisse liefert.

Außer der Schätzfunktion f(n) gibt es eine weitere Kostenfunktion, die es ermöglicht, Sätze unterschiedlicher Silbenzahl zu vergleichen: die auf Silbenzahl normierten Gesamtkosten n(n) (vgl. Abschnitt 3.3. normierte Kosten). Eine Suchstrategie, welche immer den Knoten n mit den besten normierten Kosten n(n) expandiert, geht rasch in die Tiefe des Suchbaums, da lange Sätze gegenüber kurzen bevorzugt werden. Allerdings handelt es sich hier um keine zulässige Strategie. Warum nicht ? Nehmen wir an, ein

Suchprozeß 'Depth-First', der immer dem besten n(n) nachgeht, kommt zu einem Punkt in der Lautfolge, an welchem der vorgeschaltete Klassifikator ein falsches Ergebnis geliefert hat. Aufgrund dieses Fehlers erhält ein falsches Wort f niedrigere Kosten k(f) als das wahre Wort w: $k(f) < k(w)$. Der bisherige Suchpfad habe die Gesamtkosten g(n) und eine Silbenanzahl von z. Beide Wörter w und f haben genau eine Silbe. Damit wird natürlich $n(w) = (g(n)+k(w))/(z+1)$ und $n(f) = (g(n)+k(f))/(z+1)$. Da $k(f) < k(w)$ folgt auch $n(f) < n(w)$. Der Depth-First wird also den falschen Knoten f weiter expandieren. Nehmen wir weiter an, die beiden Wörter w und f haben eine verschiedene syntaktische Stellung im Vollformenlexikon, dann wird der Depth-First unter Umständen, durch sein Syntaxwissen geleitet, einen völlig anderen Pfad im Suchbaum verfolgen. Die normierten Kosten können jedoch durchaus immer kleiner bleiben als n(w). Obwohl also ein besserer Pfad über w existiert, wird der Depth-First ein anderes Blatt als Lösung anbieten. Damit ist die Strategie nicht zulässig. Der Depth-First hat aber den Vorteil, daß er sehr rasch ein Blatt d des impliziten Suchbaums findet, wenn wir auch nicht mit Sicherheit sagen können, ob es sich um das gesuchte Zielblatt b handelt. Auf jeden Fall kann man sagen, daß die Gesamtkosten von b kleiner oder gleich sein müssen den Gesamtkosten von d:

$$g(b) =< g(d) \qquad (3.7.6)$$

Analog zur Argumentation in Abschnitt 3.7.2. läßt sich dann schließen: Alle offenen Knoten k des partiellen Suchbaums, deren Schätzkosten f(k) größer sind als g(d), können niemals zum Zielblatt b führen, denn aus

$f(k) > g(d)$ und (3.7.6) folgt

$f(k) > g(b)$ (vgl. Argumentation bei (3.7.4)).

Das bedeutet: sobald ein Blatt d gefunden wird, kann der partielle Suchbaum gelichtet werden, ohne die Optimalität zu gefährden. Diese Lichtung des Baums ist um so umfangreicher, je besser die Gesamtkosten des Blatts d an die des Zielblatts b herankommen, d.h. je ähnlicher der zunächst gefundene Satz dem besten Satz ist. Glücklicherweise zeigt sich, daß der Depth-First in der vorliegenden Anwendung bei ca. 50 % der Sätze gleich das Zielblatt b findet. Natürlich müssen jetzt die noch offenen Knoten des

Suchbaums weiter expandiert werden, um eventuell bessere Suchpfade zu finden. Das Suchverfahren geht dann analog dem A^* vor.
Die Strategiekontrolle DFA^* kann durch folgende Regeln beschrieben werden:

"Expandiere den Wurzelknoten w. Dieser enthält kein Wort und alle seine Kosten sind Null. Die Nachfolge-Knoten bilden die Menge OFFEN.

Wiederhole bis Terminalbedingung eintritt:

Expandiere immer den Knoten n aus der Menge OFFEN als nächstes, dessen normierte Kostenfunktion n(n) ein Minimum von allen offenen Knoten ist.
Entferne dann den Knoten n aus der Menge OFFEN und füge alle seine Nachfolger, Knoten und Blätter, deren Schätzkosten kleiner sind als die Gesamtkosten eines bereits gefundenen Blattes, der Menge OFFEN bei.
Wenn ein neues Blatt auftaucht, entferne alle Elemente aus OFFEN, deren Schätzkosten schlechter sind als die Gesamtkosten des neuen Blattes.

Gib das einzige Element aus OFFEN als Lösung aus."

Die Terminalbedingung des DFA^* wird zu:

"Die Menge OFFEN enthält nur noch ein Element und dieses ist nicht der Wurzelknoten w."

Diese Terminalbedingung lautet zwar ganz anders als die unter Abschnitt 3.6. definierte, erfüllt aber wegen der speziellen Art von Produktionsregel und Strategiekontrolle genau deren Bedingungen.
Wie läuft ein Suchprozeß mit DFA^* ab ?
Zunächst wird ein langer, schlanker Suchbaum rasch in die Tiefe des impliziten Suchbaums von SUCHE vordringen. Dieser kann sich im Prinzip verzweigen, tut es aber selten. Sehr bald trifft er auf ein erstes Blatt. Daraufhin verschwindet eine Anzahl von offenen Knoten längs des partiellen Baums, weil ihre Schätzkosten höher sind als die Gesamtkosten des gefundenen Blattes. War das erste Blatt nicht das beste, so wird an

einer anderen Stelle (meist nahe der Wurzel) ein zweiter, schlanker Ast in die Tiefe wachsen. Der Ablauf wiederholt sich, immer bessere Blätter werden gefunden und immer mehr offene Knoten verschwinden aus dem partiellen Suchbaum. Irgendwann wird das beste Blatt im impliziten Suchbaum gefunden. Dann werden in rascher Folge alle übrigen Knoten expandiert, aber deren Nachkommen verschwinden auch bald, da ihre Schätzkosten schnell zu groß werden. Zum Schluß verbleibt nur das beste Blatt und wird als Lösung ausgegeben.

Als einfaches Beispiel ist in Bild 3.4 die Suche nach dem gesprochenen Satz 'Er tat es in dessen Namen' dargestellt. Die Zahlen in Klammern entsprechen der Numerierung auf dem Suchpfad. Die Knoten enthalten jeweils das angehängte Wort, die normierten Kosten (NK) und Schätzkosten (SK). Der Suchprozeß beginnt in der Wurzel ROOT, welche in alle möglichen Satzanfänge expandiert wird. Der Knoten 'er' hat die niedrigsten NK von 4 und wird daher weiter expandiert (1). Alle anderen Knoten bleiben vorerst offen. Unter den Nachfolgern von 'er' hat 'er tat' die niedrigsten NK von 6 (2), aber der offene Knoten 'wenn' in der ersten Ebene liegt mit 5 darunter. Daher springt das Suchverfahren in diesen Knoten und expandiert ihn (3). Alle Nachfolger von 'wenn' (4) haben jedoch deutlich schlechtere NK als der verlassene Knoten 'er tat' und das Suchverfahren kehrt zu diesem zurück und expandiert ihn (5). Die weitere Suche (6,7, 8,9) erfolgt ohne Sprünge, bis das Blatt 'er tat es in dessen namen' erreicht wird. Der Suchprozeß erkennt, daß es sich um ein Blatt handelt, da der Knoten die richtige Silbenzahl aufweist und außerdem einen abgeschlossenen Satz enthält. Die in diesem Blatt angegebenen SK von 71 sind die echten Gesamtkosten des Satzes, da keine Restsilben mehr vorhanden sind und die Restschätzung somit zu Null wird. Es werden nun alle noch offenen Knoten gestrichen, deren SK kleiner größer sind als 71. Es bleibt nur ein Knoten übrig: 'wenn das' in der zweiten Ebene. Der Suchprozeß expandiert diesen Knoten (10) und erhält zwei Nachfolger, deren SK mit 84 bzw. 86 deutlich schlechter liegen als die Gesamtkosten des bereits gefundenen Blattes und deshalb gestrichen werden. Es bleibt allein das Blatt 'er tat es in dessen namen' als optimale Lösung.

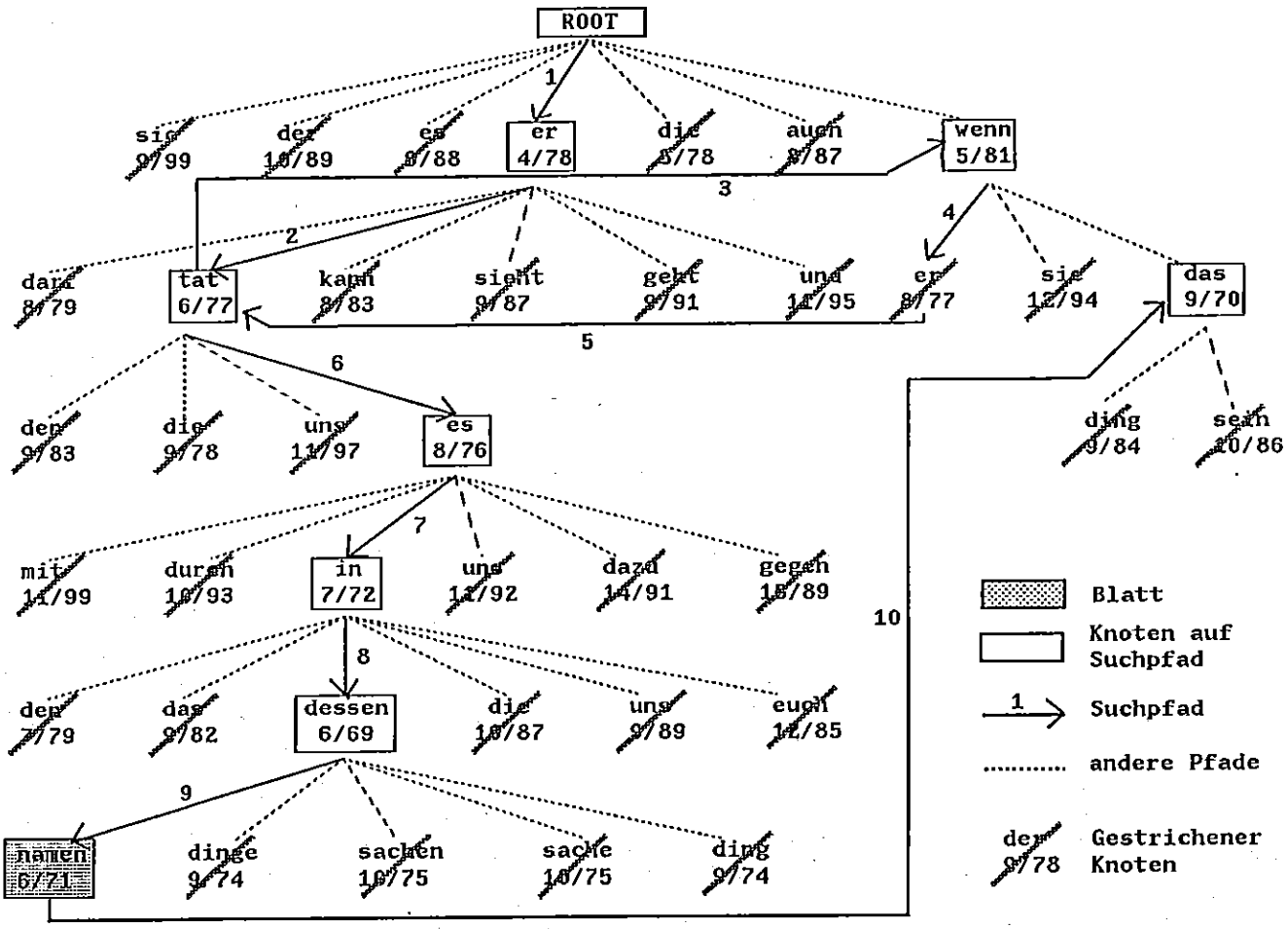


Bild 3.4 Suchpfad der Strategie DFA* im Partialbaum des Satzes "Er tat es in dessen Namen". Werte in den Knoten sind 'Normierte Kosten/Schätzkosten'.

Ist die Strategiekontrolle DFA* ein zulässiges Verfahren ? Dazu der folgende informelle Beweis:

Wir gehen davon aus, daß genau ein optimales Blatt b im impliziten Suchbaum von SUCHE existiert, dessen Gesamtkosten g(b) sind. Bevor ein Blatt gefunden ist, wird der partielle Suchbaum nicht beschnitten. Deshalb werden alle entstehenden Knoten Elemente der Menge OFFEN sein. Es muß also, wenn das erste Blatt t erzeugt wird, entweder b = t sein (Fall A) oder genau ein Knoten n_k in OFFEN enthalten sein, welcher auf dem Suchpfad von der Wurzel zu b liegt (Fall B).

Fall A: Im weiteren Suchprozeß wird nie wieder ein Blatt t_i auftauchen, denn dazu müßte $g(t_i) < g(b)$ sein (da ja sonst der DFA* dieses Blatt nicht der Menge OFFEN zufügen würde) und dies ist aber nach der Definition von b als optimales Blatt nicht möglich. Der Suchprozeß wird also

fortfahren, die Knoten n_i in OFFEN zu expandieren, wobei die Schätzkosten $f(n_i) = g(n_i) + h(n_i)$ monoton wachsen. Denn jeder Ast im Suchbaum kostet mindestens 1 (vgl. Abschnitt 3.3.), damit wächst $g(n_i)$ streng monoton und $h(n_i)$ kann minimal Null werden. Somit überschreiten irgendwann alle Knoten mit ihren Schätzkosten die Grenze g(b) und werden eliminiert. Der DFA* endet also mit b.

Fall B: Der Suchprozeß entfernt ebenfalls alle Knoten n_i, deren Schätzkosten $f(n_i) > g(t)$, aber auf keinen Fall den Knoten n_k oder dessen Nachfolger, wenn diese auf dem optimalen Suchpfad liegen. Denn es gilt:

$$f(n_k) < g(b) < g(t) \qquad (3.7.7)$$

Da für die Knoten n_j auf falschen Pfaden das gleiche gilt wie im Falle A, d.h. $f(n_j)$ wird größer als g(t) nach endlich vielen Expansionen, muß früher oder später das optimale Blatt b auftauchen. Von da an gilt das unter Fall A Gesagte, und der DFA* endet wieder in b. Die Strategiekontrolle DFA* ist also zulässig.

3.8. Zulässigkeit und Aufwand, nicht-zulässige Strategien

Im vorhergehenden Abschnitt wurde gezeigt wie sich die riesigen Baumstrukturen des reinen A* verkleinern lassen. In ungünstigen Situationen werden aber auch die Baumstrukturen des DFA* zu groß, um mit vertretbarem Aufwand an Speicherplatz bewältigt zu werden. Im Folgenden sollen daher Methoden besprochen werden, welche den Speicherbedarf des Systems SUCHE drastisch verringern, theoretisch zu nicht-zulässigen Strategien führen, aber in der Praxis die Leistung des Systems nicht beeinträchtigen.

3.8.1. Manipulation der Restschätzung h(n)

Bereits in Abschnitt 3.7.1. wurde erwähnt, daß die Genauigkeit der Restschätzung h(n) entscheidenden Einfluß auf die Güte des A* hat. Naheliegend ist es daher, dieses h(n) künstlich zu erhöhen, um dem tatsächlichen Restwert h*(n) näher zu kommen. Sei

$$h1(n) = h(n) + k * (s - z) \qquad (3.8.1)$$

z : Anzahl der Silben im bisherigen Teilsatz
s : Anzahl der Silben im gesamten Satz

eine nicht-zulässige Restschätzung, die sich aus der zulässigen Restschätzung h(n) berechnen läßt. Die Verfälschnug besteht aus einer linear mit der Silbenzahl des Restsatzes zunehmenden Multiplikation mit k. Dabei wird angenommen, daß die zulässige Restschätzung h(n) sich bei jeder Silbe des Restsatzes um einen konstanten Wert k 'irrt'. Es gibt dann entsprechend f(n) einen nicht-zulässige Schätzwert f1(n):

$$f(n) = g(n) + h(n)$$

$$f1(n) = g(n) + h1(n) \qquad (3.8.2)$$

Der beiden gemeinsame Anteil g(n) wird mit zunehmendem z immer größer, wogegen h(n) und h1(n) immer kleiner, ja für z = s sogar Null werden. Der Wert für k kann empirisch aus einer Stichprobe von Knoten auf optimalen Pfaden ermittelt werden. Näheres dazu siehe Abschnitt 7.2.4.

3.8.2. Primitive Beschneidung des partiellen Suchbaums

Unter primitiver Beschneidung wird hier das Setzen willkürlicher Grenzwerte verstanden. Z.B. läßt sich die Anzahl der Elemente in OFFEN auf einen festen oberen Wert begrenzen. Sobald dieser überschritten wird, eliminiert der Suchprozeß nach gewissen Gesichtspunkten ausgewählte Knoten. Oder die maximale Anzahl von Nachfolge-Knoten bei einer Expansion wird begrenzt: nur die ersten X besten Knoten werden überhaupt in die Menge OFFEN übernommen. Es sind noch mehr solcher Begrenzungen denkbar, allen gemeinsam ist das Problem, die richtigen Grenzwerte empirisch zu finden. Bei geschickter Wahl der Werte wird die Erkennungsleistung nicht beeinträchtigt, jedoch die Erkennungszeit deutlich verringert (s. Abschnitt 7.2.). Anhang A zeigt das Flußdiagramm eines DFA* mit drei festen Begrenzungen, der in den meisten Testläufen verwendet wurde.

3.8.3. Statistische Beschneidung des partiellen Suchbaums

Das Beschneiden der Expansionsnachfolger stellt eine Schlüsseloperation für den gesamten Suchprozeß dar. Wenn mit hoher Wahrscheinlichkeit der Erweiterungsknoten auf dem optimalen Suchpfad in den Suchbaum übernommen wird, garantiert der DFA*, daß früher oder später auch das optimale Blatt im Baum gefunden wird. Andererseits muß dringend vermieden werden, daß der Suchprozeß mit zu vielen 'falschen' Knotenerweiterungen belastet wird. Das Vorgehen im vorhergehenden Abschnitt, nämlich ein festes Limit pro Erweiterung, ist sicher keine optimale Entscheidung im Sinne statistischer Entscheidungstheorie. Denn, abgesehen von der Erfahrung dessen, der die Schwelle festlegt, wird keinerlei Information aus dem Prozeß ausgewertet. Ausgehend von einer repräsentativen Stichprobe von Sätzen ließe sich mit Hilfe der statistischen Entscheidungstheorie eine optimale Entscheidungsregel finden (Likelihood-Schwelle), die in Abhängigkeit einer aus dem bisherigen Suchprozeß gewonnenen Informationsgröße y die statistisch optimale Entscheidung trifft, wieviele und welche Knoten bei einer Expansion nach OFFEN übernommen werden ([Hau82]). Für y bietet sich z.B. an:

- die Differenz der normierten Kosten zum nächst besseren Bruderknoten.
- die Differenz der normierten Kosten von Vaterknoten zu Sohnknoten.

  etc.

Für detailiertere Informationen siehe Dokumentationsband Abschnitt 1.11. In der Praxis unserer Anwendung hat sich leider gezeigt, daß die statistische der primitiven Beschneidung nicht überlegen ist.

4. Das Problem der Syntax

Bei der Definition der Produktionsregeln des bis jetzt vorgestellten Systems SUCHE wurden keine genaueren Angaben darüber gemacht, wie der Syntaxtest von vollständigen und Teilsätzen funktionieren soll. Dies soll im folgenden Kapitel ausführlich besprochen werden. Schon hier sei festgestellt, daß für diese Aufgabe die Definition eines weiteren KI-Systems namens SYNTAX notwendig sein wird.

4.1. Formale und natürliche Sprachen

Die Syntax einer natürlichen Sprache wird durch ein festes Regelwerk mehr oder weniger eindeutig definiert. Mehr oder weniger deshalb, weil eine natürliche Sprache nicht das Produkt einer Normenkommision ist, sondern die derzeit übliche sprachliche Kommunikationsweise einer Gruppe von Menschen widerspiegelt. Trotz Duden und Rechtschreibungs-Konferenzen ändert eine lebendige Sprache ständig ihre Bestandteile, Wortschatz und Grammatik. Glücklicherweise erfolgt dabei die Entwicklung neuer Grammatik-Regeln sehr viel langsamer als die Entstehung neuer Wörter. Es muß daher möglich sein, zumindest für einen gewissen Zeitraum in der Geschichte, ein syntaktisches Regelwerk zu definieren, welches den Gebrauch von Wörtern so regelt, daß die Mehrheit der Mitglieder einer Sprachgruppe Sätze, welche mit diesem Regelwerk erstellt wurden, als syntaktisch richtig anerkennt. Solche Regelwerke existieren natürlich, größtenteils in Form von informalen 'Wenn-dann'-Formulierungen, und sind außerordentlich umfangreich, vgl. z.B. [Som88]. Kaum ein Mitglied der deutschen Sprachgemeinschaft hat dieses Regelwerk vollständig parat, aber selbst bei realistischer Reduktion auf das 'Allernötigste' bleibt eine erstaunlich große Anzahl, vor allem von wortabhängigen Sonderregelungen, z.B. daß 'schlafen' nicht im Passiv auftritt, 'Leute' nur als Mehrzahl existieren, etc. Ob ein Satz syntaktisch korrekt ist, erkennt der Mensch meistens unmittelbar, sozusagen am 'Klang' des Satzes. Da er seine Entscheidung nicht begründen kann, möchte ich dies sein implizites syntaktisches Wissen nennen. Sobald ein syntaktischer Fehler gehört wird, entsteht quasi ein Mißklang, der den Menschen veranlaßt, den gehörten oder gelesenen Satz als falsch abzulehnen. Erst in Grenzsituationen, wenn diese implizite Erkenntnisweise nur ein vages Gefühl erzeugt, setzt der Mensch seine analytischen

Fähigkeiten ein (sofern er welche hat) und versucht, durch Identifikation einzelner Satzteile und deren Beziehungen die Frage 'Richtig-oder-falsch' zu lösen. Dazu benutzt er sein explizites Syntaxwissen, welches paradoxerweise oft aus dem Lateinunterricht stammt, und versucht seine Entscheidung zu begründen. Leider ist oft in solchen Fällen diese zweite, analytische Vorgehensweise dem ersten vagen Eindruck nicht überlegen, d.h. viele Menschen entscheiden auch dann falsch, wenn sie zuerst unsicher waren und über den Satz nachgedacht haben. Dies mag auf ungenügende explizite Kenntnis der deutschen Syntax oder auf mangelndes Analysevermögen zurückzuführen sein.

Leider wissen wir bis heute wenig darüber, wie der Mensch sein implizites Wissen über Syntax einsetzt und zu solch hervoragenden Erkennungsergebnissen und -zeiten kommt. Unsere eigene Soft-und Hardware ist sozusagen (noch) nicht verstanden, wogegen die analytische Methode relativ simplen Regeln folgt, die leicht zu durchschauen sind.

Es herrscht in KI-Kreisen die weitverbreitete Meinung, daß der rasche Einsatz unseres impliziten Syntaxwissens letztendlich auf formale Regeln reduzierbar sein muß, aber diese so komplex und vielfältig sind, daß sie bis jetzt nicht durchschaut wurden ([Hof87]).

Chomsky hat gezeigt, daß jedes definierte Regelsystem einer natürlichen Sprache sich mit Hilfe eines formalen Systems abbilden läßt, wenn man einen entsprechenden Aufwand zuläßt ([Cho64]).

Ein formales System besteht mindestens aus Axiomen (Symbole, die immer (wahre) Sätze des Systems sind), Regeln und Symbolketten, die von Regeln erzeugt werden und auf die Regeln angewandt werden können. Man sagt, eine Symbolkette ist ein (wahrer) Satz des Systems, wenn sie sich aus einem oder mehreren Axiomen durch eine endliche Zahl von Regelanwendungen erzeugen (ableiten) läßt. Eine sehr anschauliche und spritzige Beschreibung formaler Systeme findet sich in [Hof87], S. 37 ff. Bildet ein formales System grammatisches Regelwerk und Wortschatz einer natürlichen Sprache ab, so kann man sagen:

> Ein Satz ist dann ein syntaktisch richtiger Satz einer natürlichen Sprache, wenn er sich in einem dieser Sprache isomorphen, formalen System als (wahrer) Satz erzeugen läßt.

Da sich ein 'Wahrhaftigkeits-Test' für Sätze eines formalen Systems immer auf einer Turingmaschine implementieren läßt, muß also eine solche existieren, die entscheiden kann, ob ein Satz einer natürliche Sprache

syntaktisch korrekt ist. So weit, so gut. Der Haken liegt natürlich am Wort 'Aufwand'. Der Aufwand, der für eine einfache deutsche Syntax getrieben werden müßte, ist immens hoch. Daher wird der Anwender bald vom hohen Roß der Theorie steigen und sich auf das Pony der Machbarkeit schwingen. Um überhaupt zu berechenbaren Dimensionen zu kommen, muß man den Anspruch der exakten Isomorphie zwischen natürlicher Sprache und formalem System aufgeben und sich mit einfacheren Sprachtypen (vgl. Chomsky-Hierarchie) begnügen. Diese haben dann natürlich eine 'Schnittmengen-Eigenschaft', d.h. die Menge der syntaktisch richtigen Sätze A und die Menge der im formalen System ableitbaren Sätze B haben eine mehr oder weniger große Schnittmenge (A ∩ B). Wünschenswert ist natürlich, daß diese möglichst groß wird, wogegen die Restmengen A/(A ∩ B) und B/(A ∩ B) möglichst klein werden sollen.

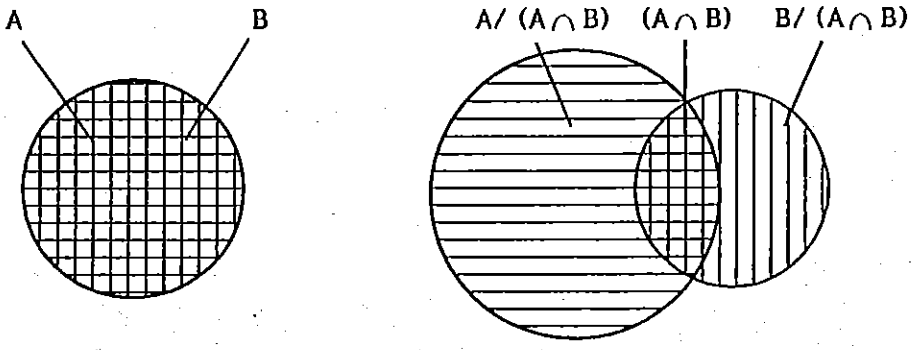


Bild 4.1. Abbildung von natürlicher Sprache auf ein formales System

(A ∩ B) ist somit die Menge aller ableitbaren, korrekten Sätze, A/(A ∩ B) die Menge aller nicht ableitbaren, korrekten Saätze und B/(A ∩ B) die Menge aller Sätze, die ableitbar sind, obwohl sie falsch sind. In der Praxis wird man zunächst die Rahmenbedingungen und Leistungen des verwendeten formalen Systems und damit zugleich die Sprachklasse in der Chomsky-Hierarchie festlegen. Dann wird man entweder aus einer allgemeinen Grammatik der betreffenden Sprache solche Regeln auswählen, die erstens für die erwartete Leistung des Systems notwendig und zweitens im Rahmen des formalen Systems in Regeln übertragbar sind, oder die notwendigen Regeln aus einer Stichprobe von Sätzen ableiten. Dieser zweite Weg

ist wahrscheinlich der einfachere und bietet zudem die Möglichkeit, die Syntax an einen begrenzten Lebensbereich, für welchen das System bestimmt ist, anzupassen.

4.2. Kontextsensitive Phrasen-Struktur-Grammatik (KPSG)

Analog dem zuletzt angesprochenen Verfahren soll nun ein formales System für unsere Anwendung erstellt werden.

Die Rahmenbedingungen seien folgende: Alleiniges Axiom ist das Symbol SATZ. Alle Sätze des Systems müssen aus diesem Axiom ableitbar sein. Die Regeln sind Ersetzungsregeln von der Form, daß links das zu ersetzende Symbol und rechts eine beliebige Kette von Symbolen stehen, welche die Ersetzung bilden, z.B. SATZ -> NP, TSFV. Es wird zwischen terminalen (Kleinschreibung) und nicht-terminalen Symbolen (Großschreibung) unterschieden. Ein terminales Symbol kann nur noch durch ein konkretes Element einer Wortgruppe ersetzt (lexikale Regel), nicht aber weiter aufgespalten werden. Eine lexikale Regel ist somit eine Regel, auf deren rechter Seite nur ein konkretes Wort des Lexikons steht. Es können beliebig viele Regeln bzw. lexikale Regeln für ein nicht-terminales bzw. terminales Symbol existieren. Zwischen allen Symbolen einer Regel, auch den Lexikoneinträgen, können beliebig viele Werte, im Folgenden Attribute genannt, ausgetauscht werden. Innerhalb einer Regel dürfen beliebige Berechnungen, Abfragen, etc. stattfinden, deren Ergebnis die Ausführung der Regel beeinflussen kann.
Unter den geschilderten Randbedingungen läßt sich eine sogenannte kontextsensitive Phrasen-Struktur-Grammatik (KPSG, engl. 'kontextsensitive definite clause grammar') realisieren. Mit ihr lassen sich Sprachen der 2. Ordnung in der Chomsky-Hierarchie darstellen. Als Grundlage für die Erstellung der nötigen Regeln dienten die 23 Testsätze im Anhang E. Diese haben zwar den Vorteil, die wichtigsten Vokale und Konsonantenfolgen der deutschen Sprache abzudecken, stellen aber nicht gerade einen geschlossenen Bereich der deutschen Syntax dar, was die Anzahl der notwendigen Regeln vergrößert. Für diese Satz-Stichprobe wurde im Rahmen von [Spi87], unter Mitwirkung von Dr. G. Thurmair (Siemens AG), eine PSG, allerdings noch ohne Attribute erstellt und von W. Kinzel in [Kin88] zur KPSG erweitert. Eine Darstellung der 32 Regeln befindet sich im Anhang B.

Da die KPSG Attribute für lexikale Einträge zuläßt, wird ein Vollformenlexikon benötigt, welches außer der Zuordnung des Wortes zu einer Wortklasse Informationen über die syntaktische Stellung (Flexion) des Wortes enthält. Das hat zur Folge, daß ein gleichlautendes Wort mehrfach vorkommt, z.B. der Artikel 'der':

```
art(der,singular,maskulinum,nominativ).
art(der,singular,femininum,dativ).
art(der,plural,maskulinum,genetiv).
usw.
```

Das für unsere Anwendung erstellte Vollformenlexikon enthält 140 Einträge, darunter 132 verschieden lautende. Es ist in Anhang C wiedergegeben. Bemerkenswert ist, daß die Attribute bis auf wenige Ausnahmen die in der Sprachwissenschaft verwendeten Flexionen, wie Genus, Numerus, Kasus und Persona, bedeuten. Eine gewünschte Erweiterung des Lexikons läßt sich daher leicht durchführen.

Wir haben jetzt ein formales System zur Verfügung, dessen ableitbare Sätze mit den oben genannten Einschränkungen syntaktisch richtige, deutsche Sätze sein sollen. Die Anzahl der in diesem System ableitbaren Sätze |B| ist schwer zu bestimmen (s. Bild 4.1.). Eine grobe Abschätzung ist mit Hilfe der Testsatz-Perplexität (TP, [Ril89]) möglich. Die TP gibt an, wieviele Satzfortführungen im Mittel bei einer Knotenexpansion auftreten, wenn eine gewisse Stichprobe von Sätzen im Suchbaum von SUCHE verfolgt wird. Mit den 23 Testsätzen (Anhang E) ergibt sich eine TP von T = 26,6. Wenn man nur Sätze mit Wortzahlen von 1 - 20 berücksichtigt (infolge der Rekursivität des formalen Systems sind durchaus auch längere, aber wegen der Rekursionseinschränkung keine unendlichen Sätze ableitbar), so ergibt sich für die Anzahl der ableitbaren Sätze |B| ungefähr:

$$|B| = \sum_{i=1}^{20} T^i = 3.03\ 10^{28} \qquad (4.2.1)$$

Das Mengenverhältnis zwischen der Menge der ableitbaren und syntaktisch richtigen Sätze ($A \cap B$) und der Menge der zwar ableitbaren, aber syntaktisch falschen Sätze $B/(A \cap B)$ ist nach empirischer Ermittlung 1 : 2.18 (s. Bild 4.1.). Damit wird die Anzahl $|(A \cap B)|$ zu

$$|(A \cap B)| = 9.53\ 10^{27}$$

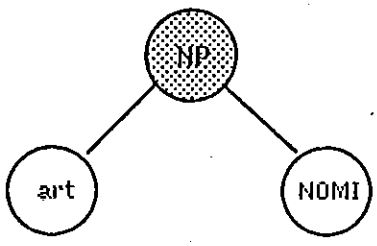
Durch das formale System KPSG können also $9.53\ 10^{27}$ syntaktisch korrekte Sätze erkannt werden. Das Verhältnis $(A \cap B)$ zu $A/(A \cap B)$ ist verständlicherweise nicht ermittelbar.

4.3. Ableitungskalkül für KPSG als KI-System

Wir haben bis jetzt ein formales System definiert, von dem wir wissen, daß ableitbare Sätze innerhalb dieses Systems mit einer gewissen Wahrscheinlichkeit auch syntaktisch richtige Sätze sind. Was noch fehlt ist ein Algorithmus, der uns zuverlässig meldet, ob ein gegebener Satz ableitbar ist oder nicht.

Der Mensch versucht eine solche Aufgabe zu lösen, indem er verschiedene Regeln auswählt, auf das Axiom und andere nicht-terminale Symbole anwendet und das Ergebnis mit dem gegebenen Satz vergleicht. Bei der Auswahl der Regeln wird er sich von seiner Erfahrung und der Kenntnis nachfolgender Regeln leiten lassen. Findet er eine Ableitung, so ist diese Methode äußerst erfolgreich. Findet er aber keine, so steht er vor der unangenehmen Aufgabe, entweder alle noch möglichen Ableitungen auszuprobieren oder sozusagen aus dem formalen System herauszuspringen und anderweitig schlüssig zu begründen, daß der Satz kein Satz des Systems sein kann. Letzteres vermag (vorläufig) nur der Mensch. Die alternative Handlungsweise aber, das Durchprobieren, kann man getrost wieder einem KI-System überlassen. Allerdings erfordert dieses System - ich möchte es SYNTAX nennen - eine ganz andere Spezifizierung als das System SUCHE, wenn auch die Grundstruktur dieselbe ist. Die Arbeitsweise von SYNTAX läßt sich analog zu Kap. 3.2. als das Absuchen eines AND/OR-Baumes interpretieren. Warum AND/OR ? Eine Ersetzungsregel der KPSG besteht aus einer Ersetzung eines nicht-terminalen Symbols durch die Verkettung mehrerer Symbole. Die Verkettung impliziert eine AND-Verknüpfung: Das linksseitige Symbol kann nur erfolgreich sein, wenn alle rechtsseitigen Symbole sich als erfolgreich erweisen.
Andererseits können innerhalb einer KPSG beliebig viele Regeln mit dem gleichen linksseitigen Symbol vorkommen. Dieses Symbol ist bereits erfolgreich, wenn nur eine dieser Regeln zum Erfolg führt. Hier ist also

eine OR-Verknüpfung ausgedrückt. Damit läßt sich ein Suchbaum definieren, ein sogenannter AND/OR-Baum, der zwei verschiedene Arten von Knoten enthält, nähmlich AND- und OR-Knoten. Ein AND-Knoten repräsentiert die Aufspaltung eines Symbols in eine Verkettung von anderen Symbolen, z.B. NP -> art,NOMI.



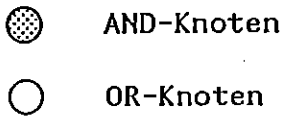


Bild 4.2. AND-Knoten NP

Ein OR-Knoten (ein Knoten im Sinne von Abschnitt 3.2.) bringt die Vielfalt eines nicht-terminalen Symbols zum Ausdruck, z.B. existieren 3 verschiedene Regeln für das Symbol ADV:

ADV -> adj. ADV -> ADV,conj,NOMI. ADV -> ADV,conj,PP.

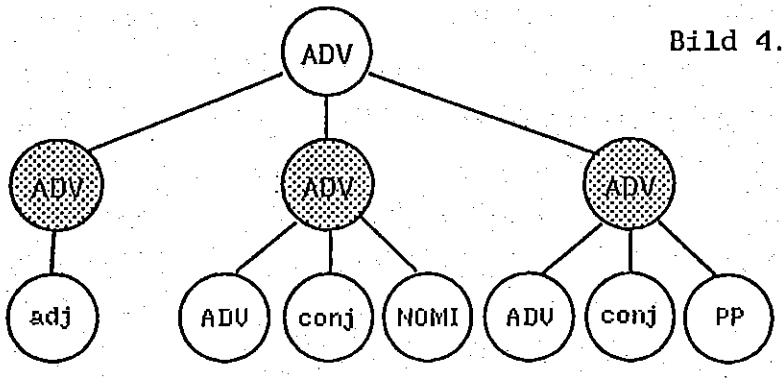
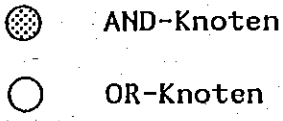


Bild 4.3. OR-Knoten ADV



Insbesondere lexikale Regeln stellen OR-Verknüpfungen dar, da es ja genügt, ein passendes Wort im Lexikon zu finden. Ein Ausschnitt des gesamten AND/OR-Baums der KPSG sowie ein Beispiel eines partiellen Suchbaums für einen bestimmten Satz befindet sich im Anhang D.
Im folgenden sei die Spezifikation des Systems SYNTAX angegeben:

Datenbasis: Wortkette, die dadurch entsteht, daß alle Blätter des AND/OR-Baums von links nach rechts verkettet werden.

Produktionsregeln: Regeln der KPSG.

Terminalbedingung: Entweder wird die Datenbasis gleich der getesteten Wortkette (erfolgreiche Beendigung), oder es läßt sich keine weitere Datenbasis mehr erzeugen (erfolglose Beendigung).

Strategiekontrolle: Da beim Anbieten eines syntaktisch falschen Satzes alle möglichen Varianten des AND/OR-Baums durchprobiert werden müssen, um sicher zu sein, daß der angebotene Satz falsch ist, wird hier anstelle einer Graphen-Suche, wie in SUCHE, ein Backtracking-Verfahren gewählt, welches sich in PROLOG sehr effektiv implementieren läßt. Beim reinen Backtracking verfügt die Strategiekontrolle über kein globales Wissen darüber, welchen von mehreren Suchpfaden aus einem OR-Knoten sie einschlagen soll. Sie versucht immer den gleichen Pfad zuerst und testet bei Mißerfolg in einer festen Reihenfolge die übrigen Pfade, bis einer von diesen erfolgreich ist.

Natürlich ist statt Backtracking auch eine Graphen-Suche denkbar, die globales Wissen für die Auswahl des besten Pfades einsetzt. Analog zu der bereits beschriebenen Strategiekontrolle (Abschnitt 3.7.) müssen dazu geeignete Kosten- und Schätzfunktionen gefunden werden. Dies gestaltet sich allerdings äußerst schwierig, da lediglich auf statistische Häufigkeit bestimmter Satzstrukturen als globale Wissensquelle zurückgegriffen werden kann. Die akustisch-phonetische Verarbeitung kann leider keine Hinweise auf die Syntax geben. Eine leichte Verbesserung läßt sich erzielen bei OR-Knoten, deren Nachfolger ein AND-Knoten ist, dessen linksseitiger Nachfolger wiederum ein Knoten mit einem terminalen Symbol ist, z.B.

NOMI -> adj,NOMI.

Da bereits im OR-Knoten NOMI durch Vergleich mit der zu testenden Wortkette das nächste zu erzeugende Blatt bekannt ist, ist über den Eintrag im Vollformenlexikon auch dessen Wortklasse yyy bekannt. Damit ist schon im OR-Knoten NOMI klar, daß der nachfolgende AND-Knoten NOMI -> adj,NOMI nur in Frage kommt, wenn yyy gleich adj ist. Das Backtracken kann AND-Knoten, auf welche diese Bedingung nicht zutrifft, einfach übergehen und spart so die Expansion eines Knotens und das sinnlose Durchsuchen einer ganzen Wortgruppe im Lexikon.

Beispiel: Der Suchprozeß sei im OR-Knoten PP angelangt. Bekannt ist, daß das nächste zu erzeugende Blatt der Wortklasse Präpositionalpronomen

(prprn) angehören muß. PP hat vier AND-Nachfolger:
PP_1 -> pr,NP. PP_2 -> prart,NOMI. PP_3 -> prprn. PP_4 -> pr,adv.
Nur PP_3 kann zum Erfolg führen, da nur in diesem AND-Knoten linksseitig das terminale Symbol prprn auftaucht. Die übrigen drei AND-Knoten können bei der Suche übergangen werden.
Das vorgestellte System SYNTAX ist nun also in der Lage, syntaktisch korrekte Sätze in Rahmen des formalen Systems zu erkennen. Durch eine leichte Modifikation wird die gleiche Leistung auch für Teilsätze erbracht (mehr dazu in Abschnitt 6.5.).

4.4. Neuformulierung der Produktionsregel in SUCHE

Wir können nun die vorläufige Definition der Produktionsregel in 3.5. vervollständigen. Folgende Aufgaben müssen erfüllt werden:
- Auswahl der Satzerweiterungen aus dem Lexikon, für die gilt:
  Das System SYNTAX entscheidet über den entstehenden Satz oder Teilsatz positiv.
- Berechnung der diversen Kosten und neuen Gesamtsilbenzahl der neuen Knoten.

Damit ist das System SUCHE und dessen Subsystem SYNTAX in allgemeiner Form vollständig beschrieben.

5. SUCHE und SYNTAX in Stichpunkten

Nachfolgend sind die erarbeiteten Definitionen für beide Systeme SUCHE und SYSTEM kurz zusammengefaßt. Diese Definitionen sollen die Ausgangsbasis für die Implementierung des Systems sein.

System SUCHE

Input:	Symbolische Lautfolge ohne Wortgrenzen, Silbenzahl Z.
Wissen:	Akustische Ähnlichkeit (Kosten), optimistische Restschätzung.
Suchbaum:	OR-Baum.
	Wurzel = leerer Satz, alle Kosten gleich Null.
	Knoten n = Teilsatz, Silbenzahl z(n), Gesamtkosten g(n), normierte Kosten n(n), Schätzkosten

f(n).

Blatt t = Knoten mit vollständigem Satz, korrekter Silbenzahl z(t)=Z.

Zielblatt b = Blatt mit geringsten Gesamtkosten g(b).

Ast = Syntaktisch richtige Erweiterung eines Teilsatzes um ein Wort.

Datenbasis: Menge von Blättern und Knoten.

Produktions-regel: Erzeugung von allen syntaktisch richtigen Nachfolgern eines Knotens der Datenbasis mit Hilfe des Subsystems SYNTAX (alle Äste, die aus einem Knoten sprießen).

Strategie-kontrolle: Zulässige Strategie Depth-First-A* (DFA*).

Terminal-bedingung: Alleiniges Blatt in Datenbasis ist Lösungssatz.

Subsystem SYNTAX

Input: Teilsatz oder Satz.

Wissen: Syntax der deutschen Sprache, repräsentiert durch kontextsensitive Phrasen-Struktur-Grammatik (KPSG).

Suchbaum: AND/OR-Baum.

Wurzel = vollständiger Satz oder Teilsatz mit speziellem Ende-Symbol.

Knoten = Nicht-terminales oder terminales Symbol.

Blatt = Lexikonwort.

Datenbasis: Verkettung der Blätter von links nach rechts.

Produktions-regel: Aufteilungsregeln der KPSG.

Strategie-kontrolle: Backtracking.

Terminal-bedingung: Datenbasis stimmt mit Wurzel überein.

6. Implementierung

In Abschnitt 3 und 4 wurden das System SUCHE und dessen Subsystem SYNTAX zur Erkennung von fließend gesprochenen Sätzen allgemein definiert. Zu Testzwecken wurden beide auf einer MicroVAX II unter VMS implementiert. Der folgende Abschnitt 6 beschäftigt sich mit den generellen Problemen der Implementierung wie : Wahl der geeigneten Sprachen und Datenstrukturen, Programmierung von Suchprozessen, Progamm-Hierarchie, Einbinden bereits vorhandener Software etc. Für Detailfragen der Programmierung sei hier auf den separaten Dokumentationsband zu dieser Arbeit verwiesen. Dort finden sich spezielle Informationen zu jedem Modul des Gesamtsystems.
Abschnitte, die die Grundkenntnis von bestimmten Programmiersprachen verlangen, sind besonders gekennzeichnet. Der Leser kann aber diese ohne weiteres überspringen und sich auf die Zusammenfassung der jeweiligen Abschnitte beschränken. Das allgemeine Verständnis wird darunter gewiß nicht leiden.

6.1. Kurze Darstellung der Software-Umgebung

Das Betriebssystem ist Micro-VMS Version 4.6. Darunter sind u.a. verfügbar: FORTRAN, C, IF-PROLOG Version 3.4. Das Page-File des Betriebssystems ist auf 12 MByte begrenzt. Das hat zur Folge, daß VMS einen Fehler generiert, sobald ein Prozeß mehr als 12 MByte Speicherplatz reserviert, obwohl der Adressierungsbereich natürlich beträchtlich größer ist. Aufrufe von FORTRAN-Subroutinen aus C und umgekehrt sind möglich. Auch Aufrufe von C-Subroutinen aus IF-PROLOG sind möglich durch Definition eines neuen IF-PROLOG-Interpreters mit Zusatzprädikaten. Der umgekehrte Weg ist versperrt: aufrufende Sprache muß IF-PROLOG sein.
Es existieren FORTRAN-Routinen, welche Vorverarbeitung, Separierung, Segmentierung und Klassifikation bis zur Symbolebene leisten. Ergebnis ist eine Symbolfolge in internationaler Lautschrift (vgl. Abschnitt 2.). Weitere FORTRAN-Routinen berechnen zu einer gewünschten Silbenposition in dieser Lautfolge nach dem Prinzip der dynamischen Programmierung die akustischen Kosten eines Lexikonwortes (vgl Abschnitt 3.3.). Alle diese Routinen sind zu einem Modul WORDSCORE zusammengefaßt. Es sind On-line-Aufnahmen, Verarbeitung von Spektraldateien (Ergebnisse der Hardware-Vorverarbeitung) oder Verarbeitung von Klassifikationsdateien (Ergebnisse

einer Klassifikation) durchführbar. Der Modul benötigt ein Wortmodell-Lexikon, welches auf den entsprechenden Klassifikatortyp trainiert ist, sowie die dazu gehörigen Verwechslungsmatrizen (Dateien lexikon.bas, akf.rsu, vok.rsu, ekf.rsu. Vgl. [Rus88], S. 155). Für den Einsatz des in WORDSCORE verfügbaren Nächsten-Nachbar-Klassifikators auf Halbsilbenbasis braucht der Modul Vergleichsmuster für die drei Silbeneinheiten Anfangskonsonatenfolge, Vokal und Endkonsonantenfolge (Dateien akf.qio, vok.qio, ekf.qio).

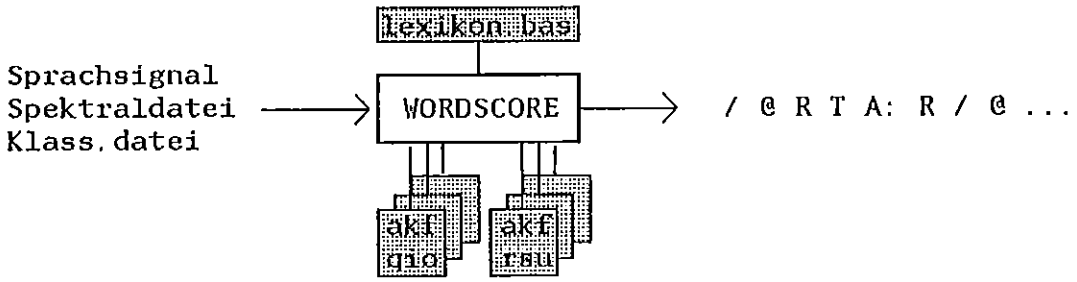


Bild 6.1 FORTRAN-Modul WORDSCORE

6.2. Prozedural oder deklarativ ?

Unter prozeduralem Vorgehen versteht man die Abarbeitung eines genau vorgegebenen Algorithmus, wobei der Programmierer in jedem Punkt des Programmablaufs genau festlegt, wie der weitere Weg des Programms verläuft. Letztendlich hat jedes Programm, das auf einer rein deterministischen Maschine läuft, eine prozedurale Vorgehensweise. Dennoch hat sich eingebürgert, auf der Ebene der höheren Programmiersprachen auch vom deklarativen Vorgehen zu sprechen. Sprachen wie C, FORTRAN, PASCAL und BASIC sind prozedural, wogegen C++ oder PROLOG als deklarativ bezeichnet werden. In einem deklarativen Programm soll nicht mehr der explizite Algorithmus zur Lösung einer Aufgabe festgelegt werden, sondern lediglich eine Faktensammlung, die das Problen ausreichend beschreibt, möglichst in einer Form, die der umgangsprachlichen Beschreibung ähnlich ist. Die tatsächliche Berechnung der Lösung soll dann duch eine geeignete, automatische Interpretation dieser Fakten erfolgen. Dabei soll und muß das Programm

nicht den besten und schnellsten Weg zur Lösung verfolgen, Hauptsache, es findet überhaupt eine Lösung, ohne daß der Anwender sich den Kopf darüber zerbricht, wie so ein Algorithmus aussehen muß. Von diesem Wunschtraum sind wir noch weit entfernt, aber dennoch bietet sich z.B. PROLOG für die Implementierung bestimmter Aufgaben an, da es eine sehr effektive Backtracking-Suche bereits eingebaut hat. Für die beiden Systeme SUCHE und SYNTAX beantwortet sich obige Frage: "Deklarativ oder prozedural ?" wie folgt:
Das System SUCHE läßt sich ohne weiteres prozedural programmieren, C wäre hier die Sprache der Wahl. Natürlich kann man auch in PROLOG eine prozedurale Vorgehensweise erzwingen und genießt dabei alle Vorteile einer sehr hochentwickelten Sprache, die den Programmierer so weit wie möglich von unangenehmer Kleinarbeit entlastet. Aber eigentlich hat SUCHE einen strikt prozeduralen Charakter.
Für das Subsystem SYNTAX dagegen bietet sich eine Programmierung in PROLOG geradezu an. Das ließe sich begründen durch die besondere Form des AND/OR-Baums, die nicht-terminalen Phrasen-Aufteilungen etc. Diese ganzen Vorteile entspringen aber einfach der Tatsache, daß PROLOG selber genau für das Problem der Syntax-Verifizierung geschrieben wurde und daher natürlich die optimale Sprache für SYNTAX darstellt. Im Abschnitt 6.5. wird dies noch klarer werden.
Die oben geschilderte ideale Kombination von C und PROLOG wurde so nicht durchgeführt, da zu Beginn der Arbeit noch nicht feststand, welche Form das System SUCHE letztendlich annehmen werde. Daher wurde der zweitbeste Weg gewählt und das System SUCHE ebenfalls in PROLOG implementiert. Der Funktionsweise tut dies natürlich keinen Abbruch, lediglich die Geschwindigkeit ließe sich wahrscheinlich bei einer Implementierung in C beträchtlich erhöhen.

6.3. Programm-Hierarchie und Dokumentation

Bevor auf die Programmierung von SUCHE und SYNTAX näher eingegegangen wird, soll hier kurz die Struktur und Hierarchie der beteiligten Module geklärt werden.

Der Modul WORDSCORE.FOR wird über eine spezielle C-Schnittstelle NP.C mit dem Standard-Interpreter von IF-PROLOG MAIN.C, sowie diversen Bibliothe-

ken zu einer neuen Interpreter-Version NP.EXE zusammengelinkt. Diese kann wie der normale PROLOG-Interpreter gestartet werden, hat aber die Möglichkeit, mit dem Modul WORDSCORE zu kommunizieren. In diesen neuen Interpreter werden verschiedene, meist kompilierte PROLOG-Module geladen, welche SUCHE und SYNTAX beinhalten, und das gesamte System mit einem bestimmten Prädikat gestartet.

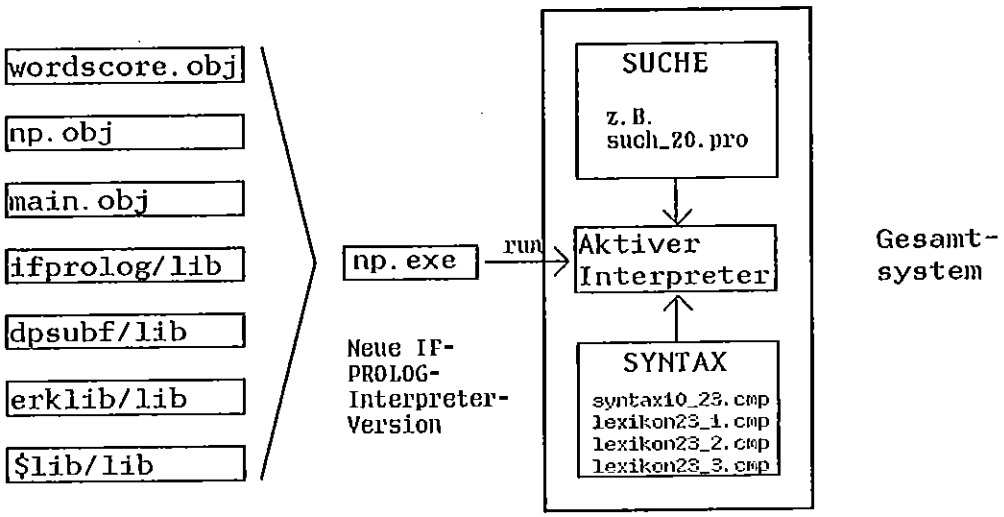


Bild 6.2 Programm-Hierarchie des Gesamtsystems

Programmtechnisch gesehen steht an der Spitze die C-Schnittstelle NP.C, denn diese wird als ausführbares Programm gestartet. Da diese aber sofort die neue PROLOG-Interpreter-Version startet, sieht der Benutzer den PROLOG-Interpreter als hierarchisch oberste Schicht, von der aus das Programm für SUCHE gestartet wird. Dieses wiederum ruft einerseits die Module von SYNTAX auf, andererseits über die C-Schnittstelle den Modul WORDSCORE.

Zu jedem einzelnen Modul existiert eine Dokumentations-Datei gleichen Namens, aber mit dem Suffix .doc im Verzeichnis vb::dua1:[sch.text.doc]. Zur Reduzierung des Speicherplatzes enthalten die meisten dieser Dokumentationen nur Änderungsangaben und verweisen sonst auf frühere Versionen. Eine vollständige Dokumentation des Moduls WORDSCORE ist nicht vorhanden, lediglich eine Funktionsbeschreibung. Für Detailfragen zur C-Schnittstelle sei auf [Ifp88] C-Interface-Manual verwiesen. Im Dokumentationsband zu dieser Arbeit ist in Abschnitt 1. eine Gesamtübersicht

über alle getesteten Versionen zu finden, im Abschnitt 2. folgen Ausdrucke der wichtigsten Module sowie deren Dokumentationsdateien.

6.4. Implementierung des Systems SUCHE

Das folgende Kapitel setzt Grundkenntnisse in PROLOG voraus. Der Leser kann aber ohne weiteres das Folgende überspringen und sich auf die Zusammenfassung am Ende des Kapitels beschränken.

Bei der Implementierung von SUCHE in PROLOG wurden bewußt typische PROLOG-Sünden begangen, um eine prozedurale Vorgehensweise zu erzwingen. Natürlich wird dies der Mächtigkeit der Sprache nicht gerecht, tut ihr aber nach meiner Erfahrung auch keinen Abbruch, denn auch prozedurales Programmieren ist in PROLOG um Vielfaches einfacher als in anderen Sprachen. Der Grundalgorithmus für SUCHE spiegelt sich in der Definition der Strategiekontrolle (vgl. Abschnitt 3.7.3., Anhang A) wider. Anstatt der sonst in PROLOG üblichen Rekursion wurde hier aber durch ein künstliches 'fail' ein iteratives Verfahren erzwungen. Im Körper von start/3 wird nach der Initialisierung des Systems durch ein repeat/0 eine Wiederholung des Prädikats suchen/5 erzwungen, bis das Argument Weiter auf 'nein' gesetzt wird. Dies kann geschehen, wenn

- die Terminalbedingung eintritt,
- keine Lösung gefunden werden kann,
- die Rechenzeit einen Grenzwert überschreitet.

Der Vorteil des iterativen Verfahrens gegenüber der Rekursion besteht darin, daß der PROLOG-Stack nach einer hohen Anzahl von Expansionen nicht überläuft. Durch das künstliche fail werden ja alle besetzten Variablen wieder freigegeben und der Stack nicht vergrößert. Jede Ausführung von suchen/5 entspricht einer Anwendung der Produktionsregel, also der Expansion eines ausgewählten Knotens in der Datenbasis. Die Datenbasis wird durch eine Liste im Prädikat list_speich/1 repräsentiert. Diese Liste enthält als Elemente strukturierte Listen, welche jeweils einen offenen Knoten des partiellen Suchbaums darstellen. Sie haben die Form:

Knoten: [SilbPos/GesKost/NormKost/SchKost,WrtNr,...,WrtNr]

SilbPos : nächste freie Silbenposition nach dem Teilsatz.

GesKost : Gesamtkosten g(n) des Knotens.
NormKost : normierte Gesamtkosten n(n) des Knotens.
SchKost : Schätzkosten f(n) des Knotens.
WrtNr...WrtNr : Teilsatz, durch Wortnummern des Lexikons bezeichnet.

Die Knoten sind innerhalb der Gesamtliste immer nach aufsteigenden normierten Kosten geordnet, dafür hat die Strategiekontrolle Sorge zu tragen. Dadurch steht immer der Knoten an erster Stelle, der als nächster expandiert werden soll.
Prädikat suchen/5 erzeugt die aktuelle Liste und übergibt sie, falls sie nicht leer ist, an suche/6, wo die eigentliche Expansion des Knotens stattfindet. Der Körper von suche/6 ist rein prozedural zu lesen. Backtracking findet hier nur statt, wenn das Programm fehlerhaft ist. Dementsprechend wird jedes Prädikat genau einmal aufgerufen und erfüllt genau eine Funktion.
anfrage/4 berechnet für alle Wörter des Lexikons die akustischen Kosten (s.o.) an der nächsten freien Silbenposition nach dem expandierten Knoten und legt die daraus entstehenden Nachfolge-Knoten in eine Liste Erweit ab. Aus dieser Liste werden durch lichte_baum_erweit/2 alle Kandidaten entfernt, deren Schätzkosten f(n) schlechter sind als die Gesamtkosten eines bereits gefundenen Blattes (Refkost). Die Elemente der Restliste GelEListe werden in syntax_test/8 auf ihre syntaktische Richtigkeit geprüft. Dabei werden sie wie folgt unterschieden: Elemente mit einer Silbenzahl, wie sie der gesuchte Satz haben muß, werden als vollständige Sätze geprüft. Ist die Prüfung erfolgreich und sind die Gesamtkosten besser als Refkost, so werden sie als Lösungskandidaten in eine gesonderte Liste übernommen. Dabei werden schlechtere Lösungskandidaten verdrängt. Schlägt der Syntax-Test fehl, so wird das entsprechende Element verworfen. Elemente mit kleinerer Silbenzahl können logischerweise nur Teilsätze sein und werden einem Teilsatz-Syntax-Test unterzogen. Bei Erfolg sind es echte Nachfolgeknoten, bei Mißerfolg werden sie ebenfalls verworfen. Hat syntax_test/8 neue Lösungskandidaten gefunden und damit Refkost neu belegt, müssen die restlichen Elemente, die den Syntax-Test überlebt haben, noch einmal gelichtet werden. Das übernimmt lichte_baum/2 und entfernt alle Knoten aus der Restliste, deren Schätzkosten f(n) schlechter sind als Refkost. Desgleichen muß in diesem Fall auch der ganze bereits existierende Partial-Baum gelichtet werden. Das geschieht

im zweiten Aufruf von lichte_baum/2, wohlgemerkt nur, wenn ein oder mehrere neue Lösungskandidaten aufgetaucht sind. Jetzt endlich wird in sort_einord/3 die Liste der echten Nachfolger-Knoten in den partiellen Baum einsortiert. Der expandierte Knoten ist in diesem bereits nicht mehr enthalten, da er im Aufruf zu suche/6 als Listenkopf abgetrennt wurde. Schließlich kann der komplette Partial-Baum noch auf eine feste Anzahl von Knoten begrenzt werden. In limitiere/5 werden nur die L3 besten Knoten, d.h. mit den besten normierten Kosten n(n), durchgelassen. Wird L3 auf z.B. 99999 gesetzt, findet keine Begrenzung statt. Der fertige Partial-Baum wird abgespeichert und der Zyklus wiederholt sich.
Noch einige Bemerkungen dazu:
Die gesamte Vorverarbeitung der Spracherkennung wird durch das Prädikat anfrage/4 ausgelöst. Bei der Expansion des Wurzelknotens wird das Argument Reset auf 1 gesetzt. Das veranlaßt den Modul WORDSCORE, sich zu initialisieren und die Vorverarbeitung bis hin zur Symbolebene durchzuführen. Bei allen weiteren Knotenexpansionen muß Reset gleich 0 sein.
Die Begrenzung des Partialbaums auf eine feste Knotenzahl ist nicht die einzige Möglichkeit, die Suche zu begrenzen. Außer L3 werden vom Prädikat start/3 noch die Parameter L1 und L2 übergeben. L1 ist die maximale Anzahl von Nachfolger-Knoten, die eine Expansion erzeugen kann. Natürlich sind dies die akustisch besten Nachfolger. L2 gibt an, welche Anzahl von Nachfolger-Kandidaten überhaupt geprüft werden sollen. Es könnte ja sein, daß zu einem Teilsatz überhaupt kein syntaktisch richtiger Nachfolger existiert. Dann würde das System aber alle 132 möglichen Wörter auf Syntax prüfen, was einen sehr hohen Rechenaufwand bedeutet. Die Wahrscheinlichkeit aber, daß an, sagen wir, 50. akustischer Stelle noch ein Nachfolger auf dem optimalen Pfad auftaucht, ist verschwindend gering. Deshalb ist es sinnvoll, nach einer bestimmten Anzahl von Prüfungen (L2) abzubrechen. Soll die Suche optimal geführt werden, müssen auch die Werte L1 und L2 auf z.B. 999 gesetzt werden.

Zusammenfassung
Das System SUCHE wird mit start(L1,L2,L3) gestartet. Die Parameter bestimmen den Grad der Begrenzung des Partial-Baums. Als eine sinnvolle Kombination hat sich L1 = 10, L2 = 20, L3 = 180 erwiesen. Soll keine Begrenzung durchgeführt werden (zulässige Suche), müssen diese genügend hoch sein. Das Programm läuft streng prozedural-iterativ ab. Rekursionen finden

nur innerhalb der einzelnen Unter-Prädikate statt. Die gesamte Vorverarbeitung bis zur Symbolebene wird zu Beginn automatisch gestartet. Alle Blätter, welche die Terminal-Bedingung erfüllen (meist nur eines !), werden ausgegeben. Der PROLOG-Kode befindet sich im Modul such.cmp, ein vollständiger Ausdruck, sowie Dokumentation dazu in Abschnitt 2. des Dokumentationsbandes zu dieser Arbeit.

6.5. Implementierung des Systems SYNTAX

Im folgenden Abschnitt wird die Implementierung eines Ableitungskalküls für das formale System KPSG (vgl. Abschnitt 4.3.) in PROLOG beschrieben. Auch hier sind einige Vorkenntnisse in PROLOG (z.B. [Snu86]) zum Verständnis unerläßlich. Am Ende des Abschnitts sind die wesentlichen Punkte zusammengefaßt.

In Anhang B und C ist das formale System KPSG wiedergegeben. Der daraus entstehende implizite AND/OR-Baum ist in seinen ersten vier Ebenen in Anhang D dargestellt. Aufgabe von SYNTAX ist es zu prüfen, ob sich zu einem gegebenen Satz oder Teilsatz eine Datenbasis erzeugen läßt, die mit diesem genau übereinstimmt. Es handelt sich also bei SYNTAX um ein Analyse-Programm ([Gia86]). In Anhang D ist ein Beispiel eines erfolgreichen Partial-Baums zum Satz "Er tat es in dessen Namen." gezeigt. Das System soll nur eine binäre Information liefern: der Input-Satz ist korrekt oder ist nicht korrekt.
Um zu zeigen, daß SYNTAX diese Aufgabe tatsächlich erfüllt, soll hier die allgemeine Struktur eines Analyse-Programms für kontextfreie Grammatiken anhand eines simplen Beispiels hergeleitet werden. Die dabei gewonnenen Regeln lassen sich auch auf kontextsensitive Grammatiken wie die KPSG anwenden. Die folgende Darstellung stützt sich weitgehend auf [Gia86], S. 79 - 95.
Gegeben sei eine Grammatik G = { S -> c, S -> aSb } und eine Sprache L(G). a,b,c seien terminale Symbole, S sei nicht-terminal und einziges Axiom von G. In einem Suchbaum ist S also immer der Wurzelknoten. Es gelte die Definition:
Eine aus terminalen Symbolen bestehende Kette ist ein Wort von L(G) genau dann, wenn sie in einem oder mehreren Schritten aus dem Axiom ableitbar ist. Z.B. ist die Kette 'c' ein Wort, denn sie ist mit der 1.Regel aus S ableitbar. Desgleichen 'aacbb': zweimal die 2. Regel, einmal die 1. Regel. Dagegen ist 'abcab' kein Wort von L(G), denn sie läßt sich durch keine Kombination von Regeln aus S erzeugen.
Eine Zeichenkette läßt sich darstellen als Übergänge zwischen verschiedenen Knoten eines Graphen. Z.B. ergibt die Kette 'aacbb' den gerichteten Graphen

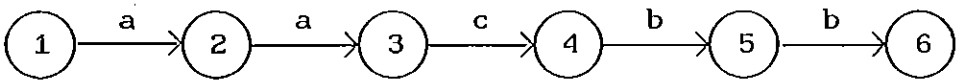


Bild 6.3 Darstellung einer Zeichenkette 'aacbb' als gerichteter Graph.

In der Sprache PROLOG läßt sich dieser Graph als Faktenmenge darstellen:

```
pfeil(a,1,2).
pfeil(a,2,3).
pfeil(c,3,4).
pfeil(b,4,5).
pfeil(b,5,6).
```

Die umgangssprachliche Interpretation von 'pfeil(a,1,2)' wäre z.B.: "Es existiert ein Pfeil von Knoten 1 nach 2 mit dem Namen a." Auf eine 'Anfrage' in Form des Prädikats 'pfeil(a,1,2).' anwortet das PROLOG-System mit 'yes'. Auf 'pfeil(b,1,2).' würde es antworten 'no'.
Auch die Regeln der Grammatik G lassen sich als Graph schreiben:

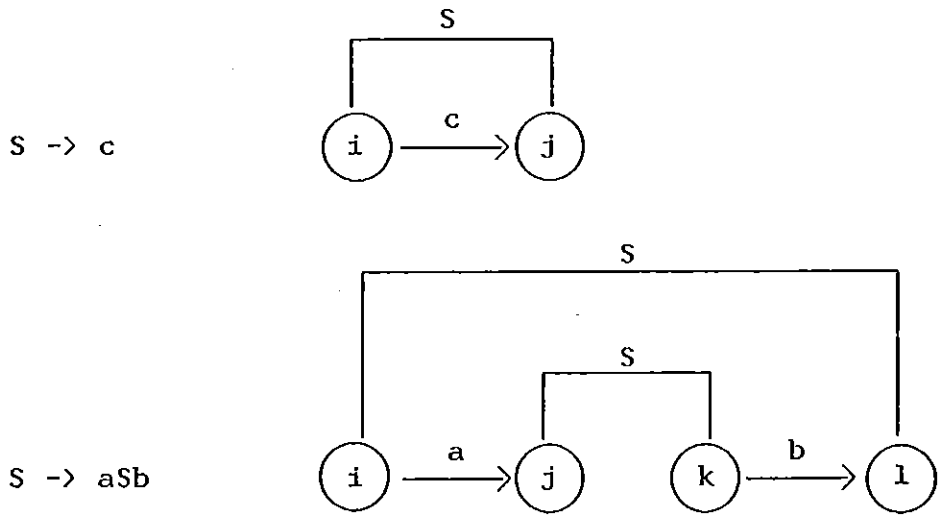


Bild 6.4 Darstellung von Grammatik-Regeln als gerichtete Graphen.

Anmerkung zur PROLOG-Notation: Das Axiom S ist in PROLOG-Notation klein zu schreiben, da Großbuchstaben in PROLOG automatisch freie Variablen - ohne semantischen Gehalt - bedeuten. Das Wurzelaxiom S hat aber durchaus semantischen Gehalt und muß daher als Atom - also kleingeschrieben - erscheinen. In PROLOG ergibt sich dann:

```
R1    pfeil(s,I,J) :-   pfeil(c,I,J).

R2    pfeil(s,I,L) :-   pfeil(a,I,J),
                          pfeil(s,J,K),
                          pfeil(b,K,L).
```

Interpretation: Das PROLOG-Prädikat 'pfeil(s,I,J).' als Anfrage bedeutet: "Gibt es einen Pfeil s von Knoten I zu Knoten J ?"
PROLOG prüft zunächst die Regel R_1 und erhält das neue Prädikat 'pfeil(c,I,J).', welches nur für I=3 und J=4 (vgl. obige Faktenmenge) erfüllt ist. Umgangssprachlich kann man sagen: PROLOG versucht den Pfeil s durch den Pfeil c zu ersetzen. Wenn ihm dies gelingt, antwortet es mit 'yes'. Im anderen Falle wird die Regel R_2 geprüft. Damit R_2 erfüllt ist, müssen drei Prädikate erfüllt sein: 'pfeil(a,I,J)','pfeil(s,J,K)' und 'pfeil(b,K,L)' Umgangssprachlich: Es ist zu prüfen, ob sich der Pfeil s von I nach J ersetzen läßt durch eine Verkettung von drei Pfeilen a, s und b von I nach J. Für die Wertepaare (I,J) = (1,6) und (2,5) ist dies möglich (vgl. obige Faktenmenge). Die Regel R_2 ist wie die korrespondierende Grammatik-Regel rekursiv und kann daher nur in Verbindung der terminalen Regel R_1 erfolgreich sein. Denn nur die Regel R_1 enthält auf der rechten Seite Prädikate (hier genau eines: 'pfeil(c,I,J)'), die allein mit Hilfe der Faktenmenge über den Graph erfolgreich sein können (hier: 'pfeil(c,3,4)'). Regel R_2 dagegen kann niemals nur aus der obigen Faktenmenge befriedigt werden, da im rechten Teil der Regel 'pfeil(s,J,K)' vorkommt und ein Faktum mit 's' als erstem Argument existiert nicht.

Beispiel:
Eine Anfrage an das System für die Kette 'aacbb' hieße in dieser Notation

```
Aufruf:    pfeil(s,1,6).
```

Regel R_1 führt nicht zum Erfolg, da das Fakt 'pfeil(c,1,6)' nicht existiert. Regel R_2 wird angewendet und ergibt das neue Prädikat 'pfeil(s,2,5)'. Regel R_1 kann auch hier nicht angewendet werden. Regel R_2 ergibt das neue Prädikat 'pfeil(s,3,4)'. Auf dieses Prädikat ist Regel R_1 erfolgreich, da 'pfeil(c,3,4)' in der Faktenmenge enthalten ist. Damit ist die Regel R_2 letztlich erfolgreich und die Antwort von PROLOG ist 'yes'.
Natürlich ist die Angabe der zu untersuchenden Kette durch ein Knotenpaar sowie die zugehörige Faktenmenge über den Graphen extrem unhandlich. Vorteil der obigen Notation ist aber, daß auch bei Mißerfolg kein Backtracking einsetzt, d.h. die Entscheidung erfolgt sehr schnell und ist unmißverständlich.
Man ersetzt nun die Knoten-Nummer in den Prädikaten rein formal durch die Zeichenkette der nachfolgenden Übergänge in Form einer PROLOG-Liste:

```
1 => [a,a,c,b,b]
2 => [a,c,b,b]
3 => [c,b,b]
4 => [b,b]
5 => [b]
6 => []                          ('=>' bedeutet 'ersetzt durch')
```

Beispiel: pfeil(c,3,4). => pfeil(c,[c,b,b],[b,b]).

Durch eine solche Ersetzung gilt immer:

```
wenn          i   => [x|U],
dann auch:    i+1 => U
```

Die Notation '[x|U]' stellt eine PROLOG-Liste mit x als erstem Element (Kopf) und der Variable U als Restliste dar.

```
Beispiel: aus     2   => [a,c,b,b]    (x=a, U=[c,b,b])
          folgt   2+1 => [c,b,b] = U
```

Angewandt auf das obige Pfeil-Prädikat 'pfeil(x,I,J)' (abgekürzt im folgenden: 'pfeil/3') bedeutet das:

```
A:    pfeil(x,I,J) => pfeil(x,[x|U],U),
```

denn die Interpretation von pfeil/3 war ja:
"Ein mit 'x' bezeichneter Pfeil geht von Knoten I zu Knoten J."
und daher muß der Knoten J als Nachfolger von I den Wert I + 1 haben.
Damit ergeben sich die neuen Regeln

```
R1'   pfeil(s,[c|U],U) :- pfeil(c,[c|U],U).
R2'   pfeil(s,[a|U],V) :- pfeil(a,[a|U],U),
                          pfeil(s,U,[b|V]),
                          pfeil(b,[b|V],V).
```

Der einzige Term von R_1' und der erste und dritte Term von R_2' sind wegen A in dieser Notation immer wahr und können daher entfallen. Auch die Faktenmenge zur Beschreibung des Graphen einer zu untersuchenden Kette entfällt, da alle Informationen jetzt in den Aufruf gepackt werden:

```
Aufruf:  pfeil(s,[a,a,c,b,b],[]).
```

Und die Regeln werden vereinfacht zu:

```
R1''   pfeil(s,[c|U],U).
R2''   pfeil(s,[a|U],V) :- pfeil(s,U,[b|V]).
```

Es fällt auf, daß das erste Argument von 'pfeil/3' ('s') praktisch nur zur Identifikation eines nicht-terminalen Symbols (hier nur S) dient. Andererseits heißen alle Regeln unserer Grammatik 'pfeil', was eigentlich keine Information mehr bedeutet. Daher kann die Identifikation 's' in den Regelkopf übernommen werden; die Regel 'pfeil(s,...)' wird zu 's(...)' :

```
R1'''   s([c|U],U).
R2'''   s([a|U],V) :- s(U,[b|V]).
```

Beispiel:
Wir wollen wieder die Kette 'aacbb' prüfen. Die Eingangskette lautet also [a,a,c,b,b], die Ausgangskette ist die leere Kette []. Der Aufruf an

unser Analyse-Programm lautet:

```
s([a,a,c,b,b],[]).
```

Regel R_1''' kann nicht angewendet werden, da die Kette im ersten Argument nicht mit 'c' beginnt. Regel R_2''' kommt zur Anwendung und führt zum neuen Prädikat (fett gedruckt):

R_2''' `s([a,a,c,b,b],[]) :- s([a,c,b,b],[b]).`

Wiederum kann Regel R_1 nicht angewendet werden. Regel R_2 kommt zur Anwendung und ergibt das neue Prädikat:

R_2''' `s([a,c,b,b],[b]) :- s([c,b,b],[b,b]).`

Jetzt greift Regel R_2''' nicht mehr, da die Kette im ersten Argument mit 'c' und nicht mehr mit 'a' beginnt. Also kommt jetzt Regel R_1 zur Anwendung:

R_1''' `s([c,b,b],[b,b]).`

Alle Prädikate sind erfüllt; das Programm antwortet mit 'yes'. Man beachte, daß die Kette 'aacba' nicht zum Erfolg geführt hätte, denn dann hätte der letzte Aufruf gelautet:

```
s([c,b,a],[b,b]).
```

Weder Regel R_1''' noch R_2''' wären darauf anwendbar; das System hätte mit 'no' geantwortet.

Bei einer Grammatik mit verschiedenen nicht-terminalen Symbolen wie der KPSG ergeben sich natürlich auch verschieden-namige Regel-Köpfe.

Anschauliche Interpretation:
Einer Start-Regel 's(X,Y)' werden eine Eingangskette X und eine Ausgangskette Y übergeben. Die Ausgangskette ist in unserem Falle immer die leere Kette []. Das Analyse-Programm versucht, die Eingangskette durch Anwendung von Regeln in die Ausgangskette überzuführen. Gelingt ihm dies, so meldet es sich

mit 'yes' und die Eingangskette ist ein Wort von L(G). Andernfalls meldet es sich mit 'no'.

Ein Analyse-Programm läßt sich aus der Notation einer kontextfreien Grammatik, wie z.B. G = { S -> c, S -> aSb }, streng formal herleiten:

1. Zu jeder Grammatik-Regel gehört genau eine PROLOG-Regel.
2. Jedes nicht-terminale Symbol geht über in einen gleichnamigen Regelkopf.
3. Jeder Regelkopf hat 2 Argumente
   a) Grammatik-Regeln, die rechts nur ein terminales Symbol haben:
      Argument 1: [x|U]
      Argument 2: U
   b) Andere Regeln:
      Argument 1: [x,...,z|U], wobei x,...,z terminale Zeichen sind, die sich vor dem ersten nicht-terminalen Symbol auf der rechten Seite der Grammatik-Regel befinden.
      Argument 2: V
4. Regelkörper
   a) Grammatik-Regeln, die rechts nur ein terminales Element haben, bekommen keinen Regelkörper.
   b) Andere Regeln:
      Der Körper enthält soviele Terme, wie die rechte Seite der Grammatik-Regel nicht-terminale Symbole enthält, und deren Köpfe haben die gleichen Namen. Der erste Term erhält als erstes Argument U. Der letzte Term erhält als zweites Argument [f,...,h|V], wobei f,...,h terminale Zeichen sind, die nach dem letzten nicht-terminalen Symbol auf der rechten Seite der Grammatik-Regel stehen. Für alle übrigen Argumente gilt:
      Argument 1 ist immer die Variable, mit der der vorhergehende Term sein zweites Argument abschließt.
      Argument 2 ist immer eine Zeichenkette [c,...,e] von terminalen Symbolen, welche nach dem dem Term entsprechenden nicht-terminalen Symbol in der Grammatik-Regel stehen, abgeschlossen mit einer beliebigen Variablen.

Beispiele:

```
S -> c            =>    s([c|U],U).
A -> abBcdeCf     =>    a([a,b|U],V) :-  b(U,[c,d,e|W]),
                                         c(W,[f|V]).
SATZ -> NP,TSFV =>      satz(U,V) :-  np(U,W),
                                      tsfv(W,V).
```

Anmerkung: In den Grammatik-Regeln sind terminale Symbole klein- und nichtterminale Symbole großgeschrieben.
Da es sich hier um eine eindeutige Abbildung handelt, lassen sich auch PROLOG-Interpreter bauen, die kontextfreie Grammatiken direkt in PROLOG-Notation umsetzen.

Der letzte Schritt zur KPSG erfolgt einfach, indem außer den beiden Listenargumenten noch beliebig viele weitere Argumente zugelassen werden, z.B.:

```
satz(U,V,Persona) :- np(U,W,Persona),
                     tsfv(W,V,Persona).
```

Hier wird zusätzlich ein Argument 'Persona' eingeführt welches für alle drei beteiligten Prädikate gleich sein muß (z.B. '1.Person'), damit die Regel erfolgreich angewendet werden kann.
Umgangssprachliche Interpretation:
"Das Symbol 'satz' kann ersetzt werden durch eine Verkettung der zwei Symbole 'np' (Nominal-Phrase) und 'tsfv' (Teilsatz mit finitem Verb), wenn beide in der Person (z.B. '1. Person') übereinstimmen."
Also : 'ich gehe' ist möglich, 'ich gehst' ist nicht möglich. Nach dem selben Muster können Numerus, Kasus, Genus etc. geprüft werden (vgl. Anhang B).

Nach dem oben beschriebenen Verfahren wurde die KPSG aus Anhang B sowie das zugehörige Vollformenlexikon aus Anhang C in PROLOG-Kode übersetzt. Der so entstandene Parser wird über das Start-Prädikat 'satz(L)' aufgerufen. Einziges Argument L ist eine Wort-Nummern-Liste, welche den zu untersuchenden Satz enthält. Um auch Teilsätze testen zu können, wird am Ende des Teilsatzes ein spezielles Symbol 'tse' (Teil-Satz-Ende) eingefügt, welches

den Parser veranlaßt, die Ableitung erfolgreich abzubrechen, sobald es an erster Stelle im zweiten Listenargument auftaucht.
Damit erfüllt das System SYNTAX genau die in Abschnitt 4.3. geforderten Aufgaben. Um nun die tatsächliche Arbeitsweise von SYNTAX anzudeuten, wird hier der Syntax-Test des Satzes 'Er tat es.' in Form einer Prädikatenliste wiedergegeben. Die Prädikatenliste enthält alle Prädikate, die der Parser für diesen speziellen Syntax-Test abarbeiten muß. Die Wörter sind ausgeschrieben, um die Lesbarkeit zu erhöhen. Vgl. dazu auch Anhang B und C.

Prädikat

```
satz([er,tat,es],[])                                   (Aufruf von SYNTAX)
np([er,tat,es],[X|Y],_,0,Num,p3,nom)                              (Regel 1)
art([er,tat,es],[tat,es,in],Num,Gen,nom)                          (Regel 4)
lex_art([er,_,Num,Gen,nom])    -> fail          ('er' ist kein Artikel)
prn([er,tat,es],[tat,es,in],Num,Pers,nom)
                              (Backtracking über Regel 1 zu Regel 5)
lex_prn([er,_,sing,p3,nom])              (Lexikoneintrag 'er' gefunden)
tat == tse   -> fail                  (kein Teilsatzende: kein Abbruch)
tsfv([tat,es],[],3,0,_,sing,p3)          (zweiter Teil von Regel 1)
fv([tat,es],[],sing,p3,_)                                -> fail
          (Regel 16 ergibt fail, da 'tat' nicht das letzte Wort ist)
tsfv([tat,es],X|Y],3,1,np,sing,p3)
                              (Backtracking über Regel 1 zu Regel 17)
fv([tat,es],[es],sing,p3,_)                                (Regel 16)
lex_fv([tat,_,sing,p3,voll])             (Lexikoneintrag 'tat' gefunden)
es == tse   -> fail                   (kein Teilsatzende: kein Abbruch)
np([es],[],_,0,_,_,_)                    (zweiter Teil von Regel 17)
art([es],[X|Y],Num,Gen,Kas)                                (Regel 4)
lex_art([es,_,Num,Gen,Kas])    -> fail          ('es' ist kein Artikel)
prn([es],[],Num,Pers,Kas)     (Backtracking über Regel 17 zu Regel 5)
lex_prn([es,_,sing,p3,akk]               (Lexikoneintrag 'es' gefunden)

=> alle geöffneten Pfade aus OR-Knoten erfolgreich beendet, SYNTAX
   antwortet mit 'yes'.
```

Zusammenfassung

Bei SYNTAX handelt es sich um ein Analyse-Programm im Sinne von [Gia86], S. 75 ff. Der Aufruf von SYNTAX erfolgt von SUCHE aus über das Prädikat 'satz(L)'. Dessen einziges Argument ist der zu untersuchende Satz oder Teilsatz in Form einer Wort-Nummern-Liste. Das Prädikat meldet 'yes', wenn es sich um einen syntaktisch korrekten Satz oder Teilsatz im Sinne des formalen Systems KPSG (Anhang B und C) handelt, im anderen Falle 'no'. Der PROLOG-Kode ist auf vier Module verteilt: Die Regeln der KPSG finden sich in syntax23_10.cmp, das Vollformenlexikon mit 132 Wörtern in lexikon23_1.cmp, lexikon23_2.cmp und lexikon23_3.cmp. Einen Ausdruck aller Module mit Dokumentationen findet sich in Abschnitt 2. des Dokumentationsbandes zu dieser Arbeit.

6.6. Vorschläge zur Beschleunigung des Suchprozesses

Der Aufwand an Speicherplatz und Rechenzeit ist bei einer zulässigen Suche für realistische Anwendungen viel zu hoch (vgl. Abschnitt 7.). Auch bei Einschränkung auf nicht-optimale Verfahren, z.B. durch Verfälschung der Restschätzungen (Abschnitt 3.8.1.) erhalten wir immer noch eine durchschnittliche Satzerkennungs-Zeit von ca. 10 min. In diesem Abschnitt sollen mehrere Vorschläge zur Verminderung von Rechenzeit und Speicherplatz gemacht werden.

a) Implementierung von SUCHE in C

Wie bereits erwähnt, läßt sich SUCHE ohne weiteres in einer prozeduralen Sprache wie C programmieren. Allerdings ist mit keinen großartigen Verbesserungen zu rechnen, da der größte Teil der Rechenzeit von SYNTAX beansprucht wird.

b) Beschleunigung des Ableitungskalküls der KPSG in SYNTAX

SYNTAX produziert für jeden zu untersuchenden Satz oder Teilsatz ausgehend von der Wurzel einen Partialbaum. Nehmen wir an, der Teilsatz 'der mensch denkt...' wurde bereits erfolgreich geprüft. Zu irgendeinem späteren Zeitpunkt verlangt SUCHE die Prüfung eines Nachfolgers dieses Knotens, z.B. 'der mensch denkt nach...'. SYNTAX beginnt nun sozusagen wieder von vorne und überprüft auch den ersten Teil 'der mensch denkt', obwohl dies früher schon einmal geschehen ist. Es muß dies auch tun, um den Kontext des Satzes wieder zu erlangen, denn nur mit Hilfe des gesamten Partial-Baums vermag es zu entscheiden, ob 'nach' eine gültige Fortsetzung ist. Im Prinzip hätte man aber den gesamten Partial-Baum von 'der mensch denkt...' abspeichern und bei der Prüfung des Nachfolgers nur an der entsprechenden Stelle im AND/OR-Baum fortfahren können. Besonders bei längeren Teilsätzen wäre dies eine enorme Rechenzeit-Ersparnis. Andererseits bedeutet dies eine drastische Vergrößerung der Datenbasis, da nun jeder offene Knoten außer den in Abschnitt 3.4. erwähnten Daten auch noch die Struktur seines Partial-Baums enthalten muß. Glücklicherweise ist dies jedoch nicht nötig. Es zeigt sich, daß es genügt, den Aufruf eines sog. Fortsetzungs-Knotens im AND/OR-Baum abzuspeichern. Dieser enthält alle Informationen, um den Ableitungsprozeß mit einer neuen Erweiterung fortzusetzen. Für die Datenbasis bedeutet dies, daß nur eine Adresse des Fortsetzungsknotens gespeichert werden muß und nicht der gesamte Partial-Baum. In Modul such_5.pro wurde dieser Vorschlag verifiziert (siehe auch Abschnitt

1.8. im Dokumentationsband zu dieser Arbeit); allerdings zeigt sich, daß sich eine Beschleunigung in PROLOG nicht erzielen läßt, da das Abspeichern der Fortsetzungsknoten in PROLOG so viel Zeit beansprucht, daß der Gewinn wieder verloren geht.

c) Beschleunigung durch bidirektionale Suchstrategie in SYNTAX

Nach der KPSG läßt sich jeder syntaktisch richtige Satz aufteilen in 2 Haupt-Phrasen:

SATZ -> NP, TSFV (1)

SATZ -> adv, TSFV (2)

SATZ -> PP, TSFV (3)

Genauso wie für SATZ läßt sich für jedes nicht-terminale Symbol der ersten 3 Regeln der KPSG eine eigene Grammatik erstellen, d.h. die nicht-terminalen Symbole NP, TSFV, PP werden zu Wurzeln von drei Unter-Grammatiken G_{NP}, G_{TSFV} und G_{PP}. Dabei seien G_{NP} und G_{PP} Grammatiken vom gleichen Typ wie die KPSG, wogegen G_{TSFV} eine inverse Grammatik darstellt, welche den Satz von hinten beginnt. Praktisch bedeutet dies, daß in allen Grammatikregeln innerhalb von G_{TSFV} die Reihenfolge der rechtsseitigen Symbole vertauscht wird, z.B. wird aus TSFV -> TSFV, NP nun TSFV -> NP, TSFV usw. Bei den Grammatiken G_{NP} und G_{PP} reduzieren sich die Anzahlen der Regeln auf 9 bzw. 13; G_{TSFV} behält alle 32 Regeln. Jede der drei Grammatiken wird nun auf einem eigenen Prozessor implementiert. Alle drei stehen unter der Koordination eines weiteren Prozessors, welcher die Ergebnisse der Sub-Prozessoren sammelt und nach obigen Regeln zu kombinieren versucht. Dies soll nur ein Beispiel sein, wie sich der Syntax-Test durch Parallelisierung beschleunigen läßt.

6.7. Bedienungsanleitung und Hinweise für Änderungen

Starten des Erkennungssystems (Eingaben sind fett gedruckt):

SET DEF VB::DUA1:[SCH.SUCH] + RET (default-directory)

NP + RET oder

NP_NOOPT + Ret (Start der entsprechenden Interpreter-Version)

Interpreter meldet sich mit einem Banner und dem Prompt '?-'

load(such). + RET

Interpreter antwortet mit 'yes'.

Vier verschiedene Aufrufe sind möglich:

- Standard-Aufruf: **start(L1,L2,L3). + RET**

- Demo-Aufruf: demo. + RET (L1=2, L2=10, L3=25)
- Bester Aufruf: optimal. + RET (L1=10, L2=20, L3=180)
- Nur Depth-First: minimal. + RET (L1=1, L2=10, L3=1)

Die Bedeutung der Parameter L1, L2 und L3 siehe in Abschnitt 6.4.
Das System fragt, wie die Vorverarbeitung geschehen soll:

- Neu sprechen (0): Direktes Besprechen, Klassifikation Nächster-Nachbar.
- Lauth.-Dat. (1): Einlesen einer Spektraldatei (= Ergebnis der Hardware-vorverarbeitung).
- Klass.-Erg. (2): Einlesen einer symbolischen Lautfolge (= Ergebnis einer Klassifikation).

Das System beginnt mit der Erkennung. Die expandierten Knoten und der Lösungssatz werden auf dem Terminal ausgegeben.
Bemerkungen:
Die Eingabe NP ist im LOGIN.COM von User SCH definiert als:

```
NP :== r [sch.such]np.exe -ss 2000k -ts 300k,
```

d.h. die Interpreter-Version np.exe wird mit einer Stack-Reservierung von 2000 KByte und einem Trace von 300 KByte aufgerufen (vgl. [Ifp88]). Je nach Klassifikator-Typ müssen die entsprechenden Dateien des Wortmodell-Lexikons im default-directory geladen sein. Es sind dies lexikon.bas, akf.rsu, vok.rsu und ekf.rsu. Im default-directory befindet sich eine Textdatei aktuell.such, welche den aktuellen Stand aller Dateien beschreibt. Dort sind auch die Adressen anderer Varianten zu finden.
Soll eine bereits vorhandene Lautheits-Datei bearbeitet werden, so muß diese vor dem Start des Systems in die Datei lauth.dat im default-directory kopiert werden. Die Spektraldateien der 23 Testsätze können auch von der PROLOG-Interpreter-Ebene mit dem Prädikat snr(SatzNr,VersNr) nach lauth.dat kopiert werden.
Das Programm bricht sich nach 7200 sec echter Zeit von selbst ab. Diese Zeitschwelle kann im Fakt deltatime/1 in such.pro festgelegt werden.
Hinweise für Änderungen:
Änderungen an den Systemen SUCHE und SYNTAX sind nur im Quell-Kode (Suffix .pro) möglich. Danach empfiehlt sich ein Interpreter-Probelauf, um Fehler schneller zu finden:
statt: load(modul). + RET, eingeben: [modul]. + RET.
Wenn der Modul fehlerfrei läuft, kompilieren mit ifcmp modul + RET (von VMS aus !).

Werden Änderungen in der C-Schnittstelle oder WORDSCORE vorgenommen, muß eine neue Interpreter-Version gelinkt werden. Der Link-Befehl lautet

```
link dua1:[sch.c]np,dua2:[IFP.c]main,dua1:[sch.c]cl/opt,
dua1:[sch.for]wordscore,dua2:[IFP.c]ifprolog/lib,
va::dub:[erkenn]dpsubf/lib,va::dub:[erkenn.erklib]erklib/lib,$lib/lib
```

und ist in der Login-Prozedur von SCH mit `link_kin` abgekürzt. Man beachte, daß sich die verschiedenen Module in den entsprechenden subdirectories befinden.

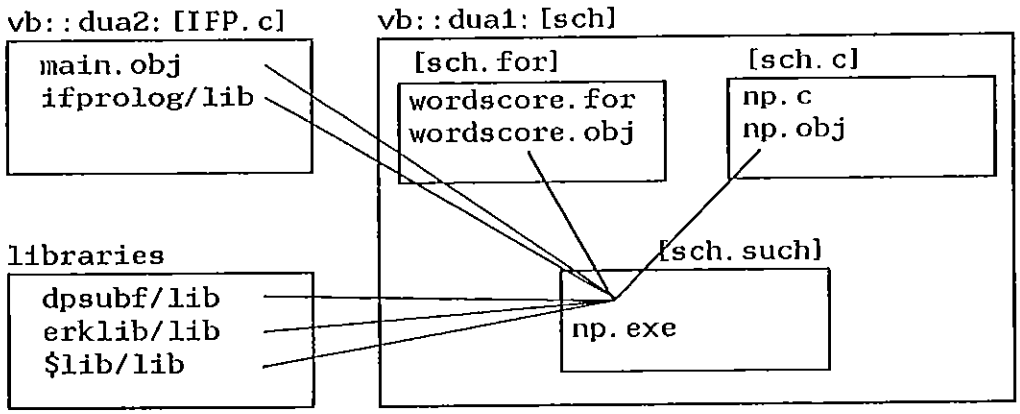


Bild 6.5 Linken der Interpreter-Version

Es empfiehlt sich, Änderungen an WORDSCORE in einem gesonderten Testprogramm vor dem Linken zu verifizieren.

7. Experimentelle Ergebnisse

In Abschnitt 7. dieser Arbeit sollen die Ergebnisse der Satzerkennung, die sich mit den beschriebenen Systemen SUCHE und SYNTAX erzielen lassen, diskutiert werden. Eine detailierte Wiedergabe aller im Rahmen dieser Arbeit gewonnenen Versuchsergebnisse findet sich in vb::dual:[sch.text.test]. In diesem subdirectory sind zu jedem getesteten System gleichnamige Ergebnis-Dateien angelegt, z.B. enthält 't_such_11.erg' alle Testergebnisse des Moduls such_11.pro.

7.1. Das Testmaterial

Das Testmaterial für die Satzerkennung bilden 23 deutsche Testsätze mit insgesamt 166 Wörtern. Darunter sind 132 unterschiedliche Wörter, welche das Lexikon des Systems bilden. Die Sätze wurden nach akustisch-phonetischen Gesichtspunkten ausgewählt und haben keinen gemeinsamen semantischen Kontext. Darüber hinaus ist die Grammatik einiger Sätze, gelinde gesagt, ungewöhnlich, obwohl keine Kommata vorkommen. Daraus resultiert eine entsprechend aufwendige KPSG.
Die 23 Testsätze wurden in 8 Versionen von einem Sprecher vorgetragen, einer Vorverarbeitung unterworfen, digitalisiert und als sogenannte Lautheits-Dateien abgespeichert. Für die weitere Verarbeitung bis hin zur Symbol-Ebene stehen zwei verschiedene Klassifikatoren zur Verfügung:

1. Klassifikation nach dem Prinzip des nächsten Nachbars mit vorgeschalteter Silben-Segmentierung ([Rus88]).
2. Klassifikation mit Hidden-Markow-Modellen ([Wei]).

Da beide Systeme nicht ohne weiteres verglichen werden können, wird hier die Leistung der verschiedenen Satzerkennungssysteme prinzipiell anhand der Klassifikations-Ergebnisse des Hidden-Markow-Klassifikators diskutiert. Sowohl der Klassifikator als auch das Wort-Modell-Lexikon und die Verwechslungsmatrizen für das Modul WORDSCORE müssen zunächst trainiert werden. Die Versionen 1 - 4 der 23 Testsätze dienen als Lernstichprobe für den Klassifikator, die Versionen 5 und 6 für das Training des Wortmodell-Lexikons und der zugehörigen Verwechslungsmatrizen. Es bleiben die Versionen 7 und 8 als echtes Testmaterial, d.h. 46 Sätze mit 332 Wörtern. Hierzu ist anzumerken, daß mit so wenig statistischem Material natürlich keine sicheren Aussagen möglich sind, der Vergleich zwischen verschiedenen Methoden der Satzerkennung ist aber

durchaus möglich.

7.2. Ergebnisse der Satzerkennung

Im folgendem sind die wichtigsten Testergebnisse der verschiedenen im Rahmen dieser Arbeit realisierten Systeme wiedergegeben.

7.2.1. Reine Tiefensuche (Depth-First)

Reine Tiefensuche bedeutet, daß die Produktionsregel des Systems SUCHE aus jedem expandierten Knoten nur eine einzige Fortsetzung erzeugt. Wie beim DFA* orientiert sich die Strategiekontrolle an den normierten Kosten der Teilsätze. Der Depth-First produziert einen einzigen Ast ohne Verzweigung von der Wurzel bis zu einem Blatt, bzw. zu einem Knoten, der sich nicht mehr expandieren läßt.
Mit dem Hidden-Markow-Klassifikator ergeben sich folgende Erkennungsraten:

Testsätze:	46
Korrekt:	23
Satz-Erkennungsrate:	50.0 %
Wort-Erkennungsrate:	24.7 %
Substitutionen:	9.6 %
Auslassungen:	65.6 %
Einfügungen:	0.6 %
Mittl. Erkennungszeit:	51 sec

Die hohe Anzahl von Auslassungen ergibt sich aus der Tatsache, daß vielfach kein Blatt gefunden wird, sondern der Suchprozeß in einem Blatt 'steckenbleibt', welches keine syntaktisch richtigen Nachfolger mehr hat. Ein solcher Satz gilt als vollständig falsch erkannt und ergibt entsprechend viele Wortauslassungen.
Auffallend ist die schnelle Verarbeitungszeit von 51 sec pro Satz, verglichen mit anderen Verfahren.
Es werden 50 % aller Sätze auf Anhieb gefunden. Dies zeigt, daß die normierten Kosten ein guter Orientierungswert für die rasche Berechnung eines Blattes sind.

7.2.2. Reiner A*-Algorithmus mit Baumbegrenzung

Die Baumbegrenzung erfolgt hier durch starre, empirisch festgelegte Grenzwerte für den Partial-Baum. Zunächst wird die maximale Anzahl von offenen Knoten im Partial-Baum auf 180 begrenzt. Damit wird erreicht, daß der beanspruchte Speicherbereich 12 MByte nicht überschreiten kann und somit das System auf einer MikroVAX II mit 12 MByte Page-File sicher läuft (vgl. Abschnitt 6.1.). Des weiteren wird die Anzahl der möglichen Expansionen aus einem offenen Knoten auf maximal 10 begrenzt, wobei die Auswahl von diesen aus den 20 akustisch besten Nachfolgern nach syntaktischen Gesichtspunkten erfolgt (vgl. Abschnitt 3.8.2.). Durch Variation dieser Konstanten in mehreren Versuchen ist sichergestellt worden, daß sich durch eine solche Beschneidung des Partial-Baums keine Verschlechterung des Ergebnisses ergibt, jedoch die Rechenzeiten erheblich verkürzt werden.
Das Suchverfahren des A* läuft analog Abschnitt 3.7.2. ab. Mit dem Hidden-Markow-Klassifikator ergeben sich folgende Erkennungsraten:

Testsätze:	46
Korrekt:	30
Satz-Erkennungsrate:	65.2 %
Wort-Erkennungsrate:	71.1 %
Substitutionen:	27.1 %
Auslassungen:	1.8 %
Einfügungen:	4.2 %
Mittl. Erkennungszeit:	9542 sek

Bevor der Suchprozeß den Partial-Baum wirksam begrenzen kann, muß er zuerst ein Blatt gefunden haben. Die Zeitspanne bis zum Auffinden des ersten Blattes ist jedoch beim A* relativ lang. War das erste Blatt nicht das optimale (ca. 60 % aller Fälle), so dauert es wieder entsprechend lange, bis ein besseres Blatt gefunden wird. Dadurch ergeben sich lange Erkennungszeiten.

7.2.3. DFA* mit Baumbegrenzung

Im Gegensatz zum reinen A* verfolgt der DFA* die Pfade mit minimalen normierten Kosten. Die Baumbegrenzung durch feste Grenzwerte erfolgt ana-

log zum A^*.
Mit dem Hidden-Markow-Klassifikator ergeben sich folgende Erkennungsraten:

Testsätze:	46
Korrekt:	31
Satz-Erkennungsrate:	67.4 %
Wort-Erkennungsrate:	91.9 %
Substitutionen:	7.2 %
Auslassungen:	0.9 %
Einfügungen:	0.9 %
Mittl. Erkennungszeit:	7945 sec

Gegenüber dem A^* zeigt sich beim DFA^* eine deutliche Verbesserung bei den Erkennungsraten und eine leichte Verminderung der Erkennungszeit.

7.2.4. DFA^* mit Verfälschung der Restschätzung

Entsprechend Abschnitt 3.8.1. wurde hier die Restschätzung h(n) mit einem konstanten Wert pro Silbe nach oben korrigiert. Der Wert wird empirisch aus einer Stichprobe von optimalen Suchpfaden gemittelt. Es ergibt sich eine Korrektur von 160 pro Silbe. Ansonsten gelten die gleichen Versuchsbedingungen wie in Abschnitt 7.2.3.
Mit dem Hidden-Markow-Klassifikator ergeben sich folgende Erkennungsraten:

Testsätze:	46
Korrekt:	34
Satzerkennungsrate:	73.9 %
Worterkennungsrate:	95.6 %
Substitutionen:	4.0 %
Auslassungen:	0.4 %
Einfügungen:	0.0 %
Mittl. Erkennungszeit:	628 sec

Wie erhofft, hat sich die Erkennungszeit deutlich verbessert (Faktor 13). Überraschend ist die Verbesserung der Erkennungsraten. Da eine Verfälschung der Restschätzung zu einer nicht-optimalen Suche führt, wäre eigentlich mit einer Verschlechterung zu rechnen. Eine genauere Untersuchung der Tester-

gebnisse im Vergleich mit Abschnitt 7.2.4. ergab für vier Testsätze folgenden Sachverhalt:
In der ersten Tiefensuche des DFA* wird auf Anhieb der tatsächlich gesprochene Satz gefunden. Trotzdem existiert aber im impliziten Suchbaum noch ein anderes, falsches Blatt mit besseren Gesamtkosten. Die erste Tiefensuche ist an dieser Alternative vorbeigegangen, da wegen einer lokal schlechten Klassifikation (meist zu Beginn) hohe normierte Kosten entstanden sind. Im DFA* ohne Verfälschung der Restschätzung wird dieses 'bessere' Blatt im weiteren Suchprozeß aufgespürt und als (falsche) Lösung ausgegeben. Mit verfälschter Restschätzung dagegen wird dieses Blatt aufgrund der nicht-optimalen Begrenzung des Partial-Baums gar nicht gefunden, und somit das akustisch zweitbeste Blatt zur (richtigen) Lösung erklärt. Ob es sich hier um ein Zufallsereignis handelt oder hier ein weiterer Hinweis auf die Güte normierter Kosten zur Orientierung im Suchverfahren vorliegt, kann nur mit Hilfe von größerem statistischen Satz-Material festgestellt werden.

7.3. Vergleich mit alternativen Methoden

7.3.1. Dynamische Programmierung ohne Syntax-Wissen

In [Ril89] wird eine 1-stufige Dynamische Programmierung zur Satzerkennung ohne jegliches Syntaxwissen beschrieben und getestet. Dieses Verfahren findet die optimale Wort-Kombination, die auf eine gegebene Lautfolge (Klassifikationsergebnis) paßt, ohne sich um grammatikalische Regeln zu kümmern. Das Verfahren wurde mit identischem Satz-Material und Klassifikation mit Hidden-Markow-Modellen getestet. Es ergeben sich folgende Erkennungsraten:

Testsätze:	46
Korrekt:	16
Satzerkennungsrate:	34.8 %
Mittl. Erkennungszeit:	11.7 sec

Die mittlere Erkennungszeit von 11.7 Sekunden pro Satz ist schon fast für Echtzeitbetrieb geeignet. Leider steht dem gegenüber eine relativ bescheidene Erkennungsrate.

7.3.2. Dynamische Programmierung mit Syntax-Graph

In [Rus88], S. 167 ff, ist ein Satzerkennungsverfahren mit Hilfe der 1-stufigen Dynamischen Programmierung beschrieben. Die Syntax-Kontrolle erfolgt dort durch einen Syntax-Graphen, vergleichbar den Syntaxgraphen von Befehlen in Programmiersprachen. Der Syntax-Graph besteht aus einem Anfangs- und Endknoten sowie Wortarten-Knoten und gerichteten Verbindungen. Jede beliebige ununterbrochene Verbindung von Anfangs- zu Endknoten soll einen syntaktisch korrekten Satz darstellen. Eigenschaften der einzelnen Wörter werden nicht zu Nachfolge-Knoten weitergegeben. Es finden keine Rekursionen innerhalb des Graphen statt. Das Verfahren ist mit dem hier vorgestellten nicht direkt vergleichbar, da es sich um unterschiedliche Satz-Materialien, unterschiedliche Klassifikatoren und unterschiedliche Grammatiken handelt. Trotzdem seien die mit diesem System erzielten Erkennungsraten hier zur Information genannt:

Testsätze:	48
Korrekt:	29
Satzerkennungsrate:	60.4 %
Worterkennungsrate:	88.5 %
Substitutionen:	11.2 %
Auslassungen:	0.5 %
Einfügungen:	1.4 %

Vorteil der 1-stufigen Dynamischen Programmierung ist die Möglichkeit sehr schneller Realisierungen auf speziellen Signal-Prozessoren. Der große Nachteil besteht darin, daß die Liste der Lexikon-Einträge einer Wortgruppe in der Abstandsmatrix nicht nur einmal vorkommt (jeder Lexikoneintrag ergibt eine Zeile in der Abstandsmatrix), sondern so oft kopiert werden muß, wie der Wortarten-Knoten im Syntax-Graphen vorkommt.
Beispiel: Die Wortgruppe 'Artikel' enthalte 10 verschieden lautende Artikel. Ohne Syntax-Überprüfung (vgl. Abschnitt 7.3.1.) führt dies zu 10 Zeilen in der Abstandmatrix des 1-stufigen DP-Algorithmus. Der Wortgruppen-Knoten 'Artikel' erscheine im Syntax-Graphen 10mal. Das ergibt 100 Zeilen in der Abstandmatrix allein für die Wortgruppe 'Artikel'.
Selbst für sehr einfache Grammatiken explodiert die Größe der Abstandsmatrix und schränkt daher die Anwendbarkeit dieses Verfahrens ein.

7.3.3. Dynamische Programmierung mit Early-Algorithmus

In [Ril89] wird eine 1-stufige Dynamische Programmierung zur Satzerkennung beschrieben, welche als Syntax-Kontrolle den sog. Early-Algorithmus verwendet. Auch dieses Verfahren verwendet nur Wortgruppen und berücksichtigt keine Syntax-Informationen - wie Kasus, Numerus und Persona - der einzelnen Wörter, basiert jedoch auf der gleichen Phrasen-Struktur-Grammatik wie die KPSG. Es wurde das gleiche Satz-Material verwendet wie in der vorliegenden Arbeit. Die Testsatz-Perplexität (vgl. Abschnitt 7.4.3.) beträgt 96.4, ist also im Vergleich zur KPSG (26.6) deutlich höher. D.h. der Early-Algorithmus arbeitet nicht so restriktiv wie die KPSG.
Mit dem in Abschnitt 7.3.1. genannten Hidden-Markow-Klassifikator und etwa vergleichbaren Expansionsbegrenzungen ergeben sich folgende Erkennungsraten:

Testsätze:	46
Korrekt:	21
Satzerkennungsrate:	45.7 %
Mittl. Erkennungszeit:	230 sec

Die Erkennungszeit ist um den Faktor 2 niedriger als beim besten Suchprozeß nach DFA*. Dafür bewirkt die niedrige Restriktivität eine deutlich geringere Satzerkennungsrate.

7.4. Tabellarischer Vergleich aller Verfahren

Verfahren	Satzerkennungsrate	Worterkennungsrate	Substitutionen	Auslassungen	Einfügungen	Erkennungszeiten
Reiner Depth-First	50.0 %	24.7 %	9.6 %	65.6 %	0.6 %	51 sek
Reiner A*-Algorithmus	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]
DFA* mit Baumbegrenzung	67.4 %	91.9 %	7.2 %	0.9 %	0.9 %	7945 sek
DFA* m. Verfälsch. der Restsch.	73.9 %	95.6 %	4.0 %	0.4 %	0.0 %	628 sek
Dynamische Prog. ohne Syntax	34.8 %	-	-	-	-	11.7 sek
Dynamische Prog. m. Early-Alg.	45.7 %	-	-	-	-	230 sek
Dynam. Prog. mit Syntax-Graph	60.4 %	88.5 %	11.2 %	0.5 %	1.4 %	-

8. Gesamtbeurteilung

Das im Rahmen dieser Arbeit entwickelte und implementierte Satzerkennungs-System zeigt gegenüber vergleichbaren Verfahren eine deutliche Verbesserung der Satz- und Worterkennungsrate. Es ist zu erwarten, daß diese sich bei Verwendung eines besser trainierten Klassifikators und Wortmodell-Lexikons noch deutlich erhöhen.

Dem gegenüber steht eine für Echtzeitanwendungen viel zu hohe Erkennungszeit von ca. 10 min pro Satz. Mit den in Abschnitt 6.6. gemachten Vorschlägen zur Beschleunigung des Verfahrens - Implementierung des Systems SUCHE in C, Parallelisierung des Syntax-Tests in SYNTAX - ist eine Beschleunigung von 2 Größenordnungen denkbar. Damit wäre die mittlere Erkennungszeit pro Satz im Bereich von einigen Sekunden. Auch bei der Erkennungszeit ist eine Verbesserung zu erwarten, wenn mehr statistisches Sprachmaterial zum Training des Systems zu Verfügung steht und folglich die Ergebnisse der Klassifikation besser werden. Das hier vorgestellte System 'denkt' um so länger nach, je schlechter das Klassifikationsergebnis der unteren Ebenen ist. Dies läßt sich zeigen, indem man dem Satzerkennungs-System Sätze in echter Lautschrift anbietet. Die Erkennungszeit liegt dann meist im Bereich von einer Minute oder weniger.

Anhand der Systeme SUCHE und SYNTAX wurde gezeigt, daß sich Methoden der Künstlichen Intelligenz - insbesondere Baumsuchverfahren - erfolgreich für die automatische Erkennung von fließender Sprache einsetzen lassen. Im Zuge der Weiterentwicklung der Datenverarbeitungstechnik hin zu schnelleren Prozessoren und größeren Speichern werden solche Systeme bald für den praktischen Einsatz nutzbar sein.

Schwierigkeiten bestehen vor allem noch in der Formulierung einer geeigneten, streng formalen Grammatik, welche einen möglichst großen Teil der gesprochenen Sprache erfassen soll. Bei einer weiteren Verfeinerung der Grammatiken ist jedoch zu erwarten, daß man ohne semantisches Hintergrundwissen nicht mehr auskommt. Für einfache Aufgabenstellungen dagegen (z.B. Auskunftsysteme über einen streng begrenzten Bereich des Lebens) ist sogar eine einfachere Grammatik denkbar, als in dieser Arbeit verwendet wurde.

Literaturverzeichnis

Cho64 N. Chomsky: The logical basis of linguistic theory
aus: H.C. Lunt (Hrsg.), Proceedings of the ninth International Congress of Linguistics. S. 914-978,
The Hague, 1964.

Gia86 F.Giannesini, H. Kanoui, R. Pasero, M. van Caneghem: Prolog
Addison-Wesley Deutschland, 1986.

Gro79 B.J. Grosz: Utterance and objective: issues in natural language processing
in IJCAI-6, pp 1067 - 1076, 1979.

Hau89 G. Hauske: Einführung in die Kybernetic, Skriptum S. 5 ff
Lehrstuhl für Nachrichtentechnik TUM, 1982.

Hof87 D. Hofstatter: Gödel, Escher, Bach
Klett-Cotta, 1987.

Ifp88 IFPROLOG Manual Version 3.4.0,
Interface Computer GmbH, 1988.

Kel89 Albert Keller: Sprachphilosophie,
Verlag Karl Alber Freiburg Muenchen, 1989.

Kin88 Winfried Kinzel: Syntaxsteuerung für die automatische Spracherkennung unter Verwendung von PROLOG,
Diplomarbeit am Lehrstuhl für Datenverarbeitungstechnik,
Technische Universität München, 1988.

Kuh89 Thomas Kuhn: Eine Suchstrategie zur kontextfreien Analyse von Worthypothesen,
Diplomarbeit am Lehrstuhl für Informatik 5, Institut für mathematische Maschinen und Datenverarbeitung,
Friedrich-Alexander-Universität Erlangen-Nürnberg, 1989.

Nil71 Nils J. Nilson: Problem-Solving Methods in Artificial Intelligence,

McGraw-Hill Book Company, New York, 1971.

Nil82 Nils J. Nilson: Principles of Artificial Intelligence, Springer-Verlag Berlin Heidelberg New York, 1982.

Ril89 Bernhard Rilk: Satzerkennung mit syntaxgesteuerter Dynamischer Programmierung, Diplomarbeit am Lehrstuhl für Datenverarbeitung, Technische Universität München, 1989.

Rus88 Dr.Ing. habil. G. Ruske: Automatische Spracherkennung, R. Oldenbourg Verlag München Wien, 1988.

Spi87 Burkhard Spiegel: Portierung des C-PROLOG auf CADMUS und VAX und Implementierung eines natürlichsprachlichen Parsers, Diplomarbeit am Lehrstuhl für Datenverarbeitung, Technische Universität München, 1987.

Snu86 Peter Schnupp: PROLOG, Einführung in die Programmier-praxis, Carl Hanser Verlag München Wien, 1986.

Sha66 D. Shapiro: Algorithms for the Solution of the Optimal Cost Traveling Salesman Problem, Sc.D. thesis, Washington University, St. Louis, 1966.

Som88 Karl-Ernst Sommerfeldt (Hrsg.): Einführung in die Grammatik der deutschen Gegenwartssprache, VEB Bibliographisches Institut Leipzig, 1988

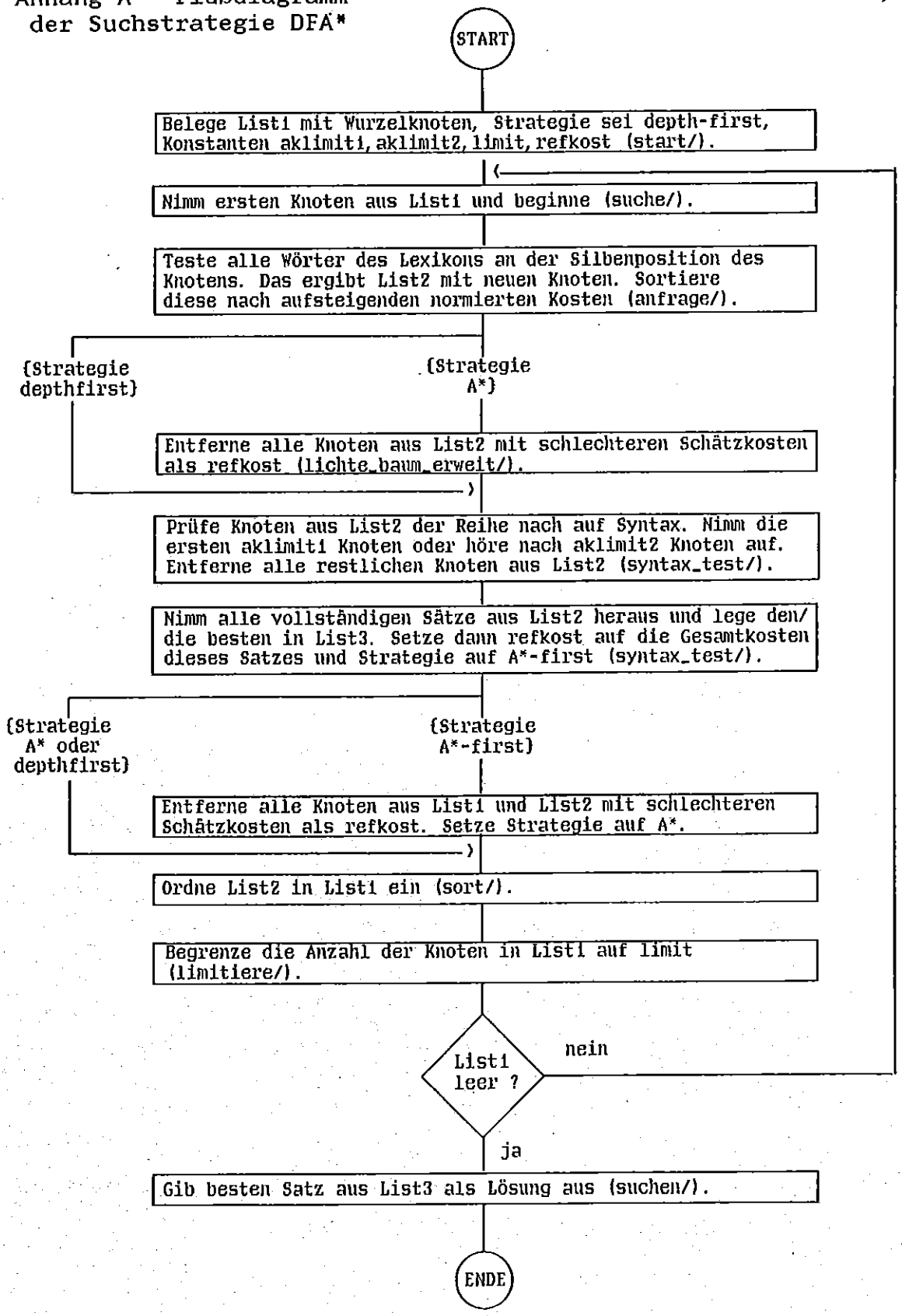
Win72 T. Winograd: Understanding Natural Language Academic Press, New York, 1972.

Wei W. Weigel: Silbenorientierte Erkennung fließend gesprochener Sprache unter Berücksichtigung von Segmentierungsfehlern und Koartikulation (Arbeitstitel), Dissertation, Lehrstuhl für Datenverarbeitung, Technische Universität München, in Vorbereitung.

Verzeichnis der wichtigsten Abkürzungen

A	Menge der syntaktisch richtigen, deutsche Sätze
|A|	Anzahl der Elemente von A
adj	Adjektiv, terminales Symbol der KPSG
ADV	Adverb, nicht-terminales Symbol der KPSG
art	Artikel, terminales Symbol der KPSG
A*	A*-Algorithmus
B	Menge der mit SYNTAX ableitbaren Sätze
|B|	Anzahl der Elemente von B
conj	Konjunktion, terminales Symbol der KPSG
DFA*	Depth-First-A*-Algorithmus
f(n)	Schätzkosten des Teilsatzes im Knoten n
g(n)	Gesamtkosten des Teilsatzes im Knoten n
h(n)	Restschätzung von Knoten n bis zum besten Blatt
KPSG	Kontextsensitive Phrasen-Struktur-Grammatik
k(X)	Kosten der Einheit X
KI	Künstliche Intelligenz
NK	Normierte Kosten
NOMI	Nominal-Ausdruck, nicht-terminales Symbol der KPSG
NP	Nominal-Phrase, nicht-terminales Symbol der KPSG
n(n)	auf Silbenzahl normierte Gesamtkosten des Teilsatzes in Knoten n
pr	Präposition, terminales Symbol der KPSG
prart	Präpositional-Artikel, terminales Symbol der KPSG
PP	Präpositional-Phrase, nicht-terminales Symbol der KPSG
prprn	Präpositional-Pronomen, terminales Symbol der KPSG
PSG	Phrasen-Struktur-Grammatik
SATZ	Axiom der KPSG, nicht-terminales Symbol
SK	Schätzkosten
TP	Testsatz-Perplexität
TSFV	Teilsatz mit finitem Verb, nicht-terminales Symbol der KPSG

Anhang A - Flußdiagramm der Suchstrategie DFA*



Anhang B – Kontextsensitive Phrasen-Struktur-Grammatik

```
1   SATZ -> NP(S=0,Num,Pers,Kas=nom) , TSFV(Itermax=3,N=0,Num,Pers)

2   SATZ -> adv() , TSFV(Itermax=3,N=0)

3   SATZ -> PP() , TSFV(Itermax=3,N=0)

4   NP(S,Num,Pers=p3,Kas)   -> art(Num,Gen,Kas) ,
                               NOMI(S,Num,Gen,Kas)

5   NP(Num,Gen,Kas)   -> prn(Num,Gen,Kas)

6   NP(S,Num,Pers=p3,Kas) -> NOMI(S,Num,Gen,Kas,Count)
        entweder: Count=nein  oder  Num=plur

7   NP(S=0,Num,Pers=p3,Kas=nom) -> npmod() ,
                                   NP(C=2,S=1,Num,Pers=p3,nom)
        keine Rekursion auf 7 oder 8

8   NP(C=1,S=0,Num,Pers=p3,Kas) -> NP(C=0,Kas=gen) ,
                                   NOMI(S=0,Num,Kas)
        keine Rekursion auf 8

9   PP(S) -> pr(Kas) , NP(S,Kas)

10  PP(S) -> prart(Num,Gen,Kas) , NOMI(S,Num,Gen,Kas)

11  PP() -> prprn()

12  PP(S=0) -> pr() , adv(S=0)

13  adv(S=1) -> adj(Kas=adj)

14  adv(S=1) -> adv(S=0) , conj() , NOMI(C=0)
        keine Rekursion auf 13, 14, 15

15  adv(S=1) -> adv(S=0) , conj(), PP(S=1)
```

keine Rekursion auf 13, 14, 15

16 TSFV(Num,Pers) -> fv(Num,Pers)

17 TSFV(Itermax,N,Num,Pers) -> TSFV(Itermax,N+1,Num,Pers),
NP(S=0)
N+1 <= Itermax

18 TSFV(N=0,Num,Pers) -> fv(Num,Pers,Voll=zs) ,
TSNFV(Itermax=3,N=0)

19 TSFV(Itermax,N,Num,Pers) -> TSFV(Itermax,N+1,Num,Pers) , PP()
Max. 1 Rekursion auf 19, Itermax <= N+1

20 TSFV(Itermax,N,Num,Pers) -> TSFV(Itermax,N+1,Num,Pers) ,
adj(Kas=adj)
Max. 1 Rekursion auf 20, Itermax <= N+1

21 TSFV(Itermax,N,Num,Pers) -> TSFV(Itermax,N+1,Num,Pers) ,
adv(S=0)
Max. 1 Rekursion auf 21, Itermax <= N+1

22 TSFV(Itermax,N=0,Num,Pers) -> TSFV(Itermax,N=0,Num,Pers) ,
vz()
Max. 1 Rekursion auf 22

23 TSNFV(Itermax,N) -> NP(S=1) , TSNFV(Itermax,N+1)
Itermax <= N+1

24 TSNFV(Itermax,N) -> adv() , TSNFV(Itermax,N+1)
Itermax <= N+1

25 TSNFV() -> NFV()

26 TSNFV(Itermax,N) -> adj(Kas=adj) , TSNFV(Itermax,N+1)
Itermax <= N+1

27 TSNFV(Itermax,N) -> PP() , TSNFV(Itermax,N+1)
Itermax <= N+1

```
28  NFV() -> infz() , NFV()

29  NOMI(Num,Gen,Kas,Count) -> nomen(Num,Gen,Kas,Count)

30  NOMI(C=0,S,Num,Gen,Kas,Count) -> adj(Num,Gen,Kas) ,
                                     NOMI(C=1,S,Num,Gen,Kas,Count)

31  NOMI(S,Num,Gen,Kas,Count) -> nomen(Num,Gen,Kas,Count),
                                 NP(S=10,Kas=gen)
        S <> 10

32  NOMI(S,Num,Gen,Kas,Count) -> nomen(Num,Gen,Kas,Count),
                                     PP(S=10)
        S <> 10
```

Sonderregelungen:

1. adv ist sowohl terminales (adv(lex)) als auch nicht terminales Symbol (vgl. Regeln 13, 14 und 15). Die nicht terminalen Regeln können jedoch nicht auf sich selbst angewandt werden:

Beispiele:

```
adv -> adv(lex).        Terminale Regel

adv -> adj(Kas=adj),
adj -> adj(lex).        Regel 13: Ersetzung Adverb durch
                                  adverbiales Adjektiv

adv -> adv , conj , NOMI(C=0),
adv -> adv(lex),
conj -> conj(lex),
NOMI(C=0) -> < Regeln 29 - 32 >.
                        Regel 14: Ersetzung Adverb durch Folge
                                  Adverb, Konjunktion, Nominal
```

```
adv -> adv , conj , PP(S=1),
adv -> adv(lex),
conj -> conj(lex),
PP(S=1) -> < Regeln 9, 10, 11 >.
```

Regel 15: Ersetzung Adverb durch Folge Adverb, Konjunktion und Präpositionalphrase

2. Rekursionseinschränkungen:

Regel 4 : z.B. 4, 31 (o. 32), 4, 31 (o. 32) nicht möglich
Regel 6 : z.B. 6, 31 (o. 32), 6, 31 (o. 32) nicht möglich

u.s.w.

Anhang C – Vollformenlexikon für KPSG

```
adj,bestimmt,_,_,adj
adj,ernst,_,_,adj
adj,ersten,sing,fem,dat
adj,franzoesischen,sing,mas,gen
adj,ganz,_,_,adj
adj,gluecklich,_,_,adj
adj,groesste,sing,mas,nom
adj,kleinen,plur,fem,nom
adj,kurz,_,_,adj
adj,letzte,sing,fem,nom
adj,naechsten,sing,fem,dat
adj,schnell,_,_,adj
adj,schwer,_,_,adj
adj,sicher,_,_,adj

adv,allerdings
adv,also
adv,anders
adv,daraufhin
adv,deshalb
adv,durchaus
adv,gern
adv,insbesondere
adv,jetzt
adv,kaum
adv,links
adv,nicht
adv,sogleich
adv,spaeter
adv,uebrigens
adv,wirklich
adv,zunaechst

art,aller,plur,neut,gen
art,den,plur,mas,dat
art,den,sing,mas,akk
```

art,[illegible],gen
art,[illegible],dat
art,[illegible],nom
art,[illegible],gen
art,[illegible],mas,gen
art,[illegible]
art,die,sing,fem,nom
art,die,sing,fem,akk
art,die,plur,_,nom
art,diese,sing,fem,akk
art,diese,plur,_,nom
art,einen,sing,mas,akk
art,einige,plur,fem,akk
art,fuenf,plur,neut,nom
art,jene,sing,fem,akk
art,sein,sing,mas,nom
art,seiner,sing,fem,gen

conj,als

fv,bleiben,plur,p3,voll
fv,blieb,sing,p3,voll
fv,bringt,sing,p3,voll
fv,darf,sing,p3,zs
fv,duerfen,plur,p3,zs
fv,erscheint,sing,p3,voll
fv,folgt,sing,p3,voll
fv,hat,sing,p3,zs
fv,ist,sing,p3,zs
fv,kannst,sing,p2,zs
fv,koennen,plur,p3,zs
fv,macht,sing,p3,voll
fv,scheint,sing,p3,zs
fv,sprach,sing,p3,voll
fv,tat,sing,p3,voll
fv,vertrauen,plur,p3,voll
fv,war,sing,p3,voll
fv,waren,plur,p3,zs
fv,zeigt,sing,p3,vz

fv,ziehen,plur,p3,voll

infz,zu

nfv,entfernt
nfv,enthalten
nfv,erklaeren
nfv,geblieben
nfv,gehabt
nfv,geworden
nfv,haben
nfv,reden
nfv,sein

nomen,anfang,sing,mas,nom,ja
nomen,anspruch,sing,mas,nom,ja
nomen,armen,plur,mas,gen,ja
nomen,art,sing,fem,nom,ja
nomen,ausdruck,sing,mas,akk,nein
nomen,bank,sing,fem,dat,ja
nomen,beziehung,sing,fem,akk,ja
nomen,dinge,plur,neut,gen,ja
nomen,dritten,plur,mas,nom,ja
nomen,einfluss,sing,mas,akk,nein
nomen,erfolg,sing,mas,akk,nein
nomen,fluegeln,plur,mas,dat,ja
nomen,freund,sing,mas,nom,ja
nomen,geld,sing,neut,nom,nein
nomen,gestalt,sing,fem,_,ja
nomen,haelfte,sing,fem,_,ja
nomen,kampf,sing,mas,akk,ja
nomen,koenigs,sing,mas,gen,ja
nomen,kopf,sing,mas,nom,ja
nomen,kraefte,plur,fem,nom,ja
nomen,kuenstler,sing,mas,nom,ja
nomen,massregeln,plur,fem,nom,ja
nomen,mensch,sing,mas,nom,ja
nomen,namen,sing,mas,dat,ja
nomen,neue,sing,neut,akk,ja

```
nomen,paragraph,sing,mas,nom,ja
nomen,pflicht,sing,fem,akk,ja
nomen,platz,sing,mas,akk,ja
nomen,prozent,plur,neut,nom,ja
nomen,regierung,sing,fem,akk,ja
nomen,sache,sing,fem,_,ja
nomen,schlacht,sing,fem,_,ja
nomen,schrift,sing,fem,_,ja
nomen,strasse,sing,fem,dat,ja
nomen,stunden,plur,fem,akk,ja
nomen,taetigkeit,sing,fem,_,ja
nomen,truppen,plur,fem,nom,ja
nomen,wechsel,plur,mas,gen,ja
nomen,wuenschen,plur,mas,dat,ja
nomen,zukunft,sing,fem,nom,ja
nomen,zweiten,plur,mas,dat,ja

npmod,erst
npmod,selbst

pr,auf,akk
pr,auf,dat
pr,aus,dat
pr,gegen,akk
pr,in,akk
pr,in,dat
pr,mit,dat
pr,nach,dat

prart,aufs,sing,neut,akk

prn,du,sing,p2,nom
prn,er,sing,p3,nom
prn,es,sing,p3,akk
prn,nichts,sing,p3,nom
prn,sich,_,p3,dat
prn,sie,plur,p3,nom
prn,uns,plur,p1,dat
```

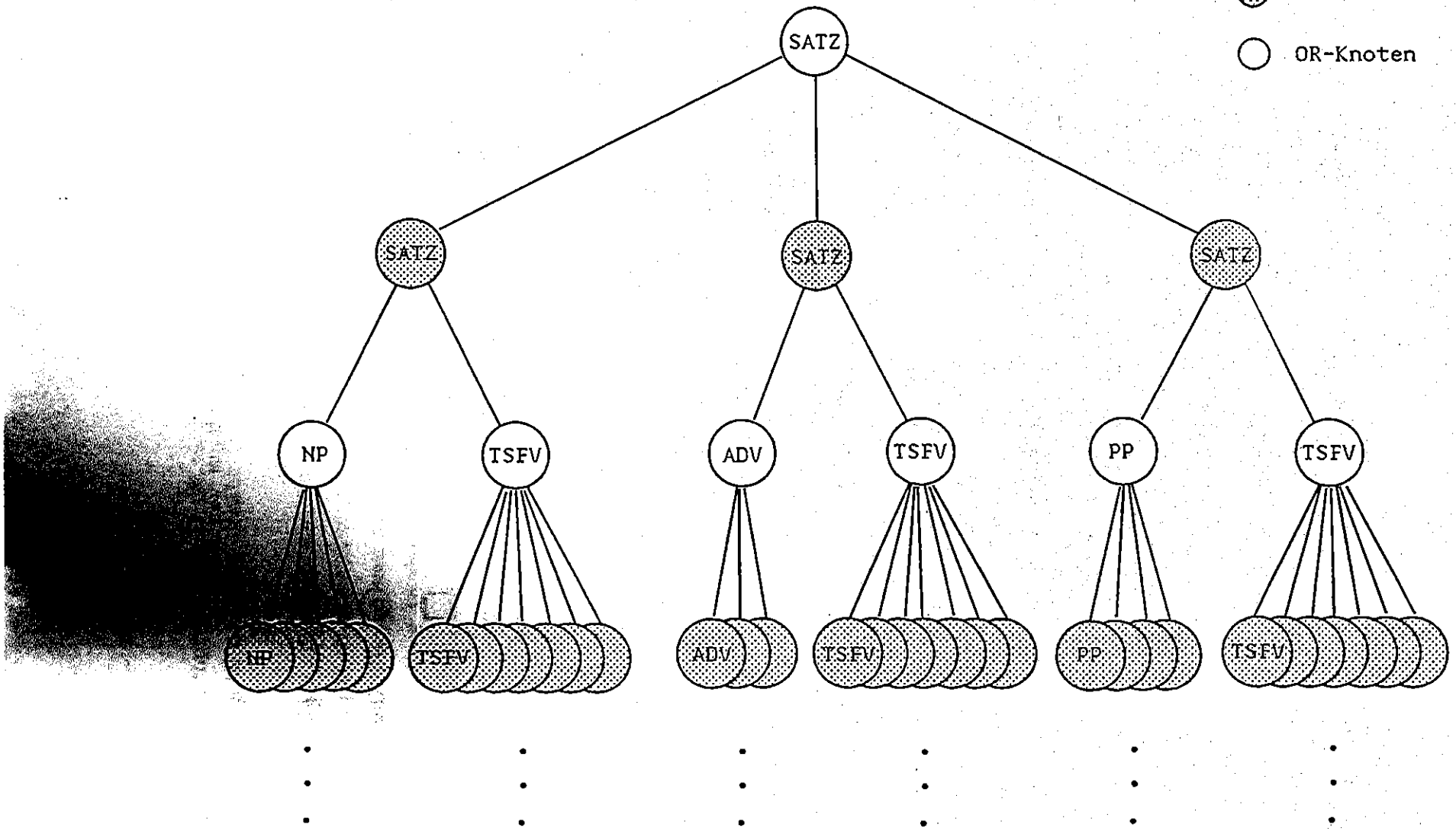
prprn,dagegen

vz,auf

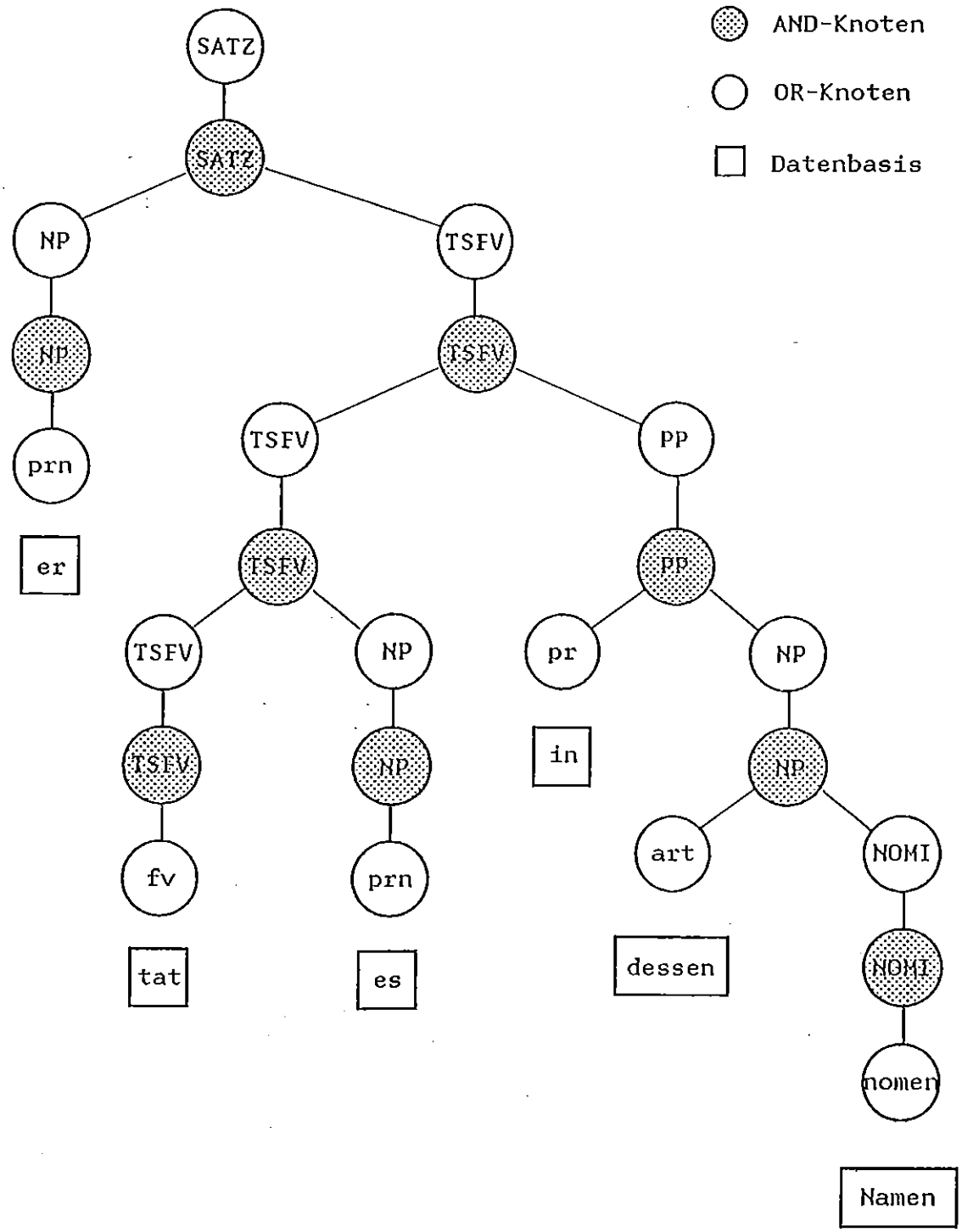
Abkürzungen

adj	Adjektiv	adv	Adverb
art	Artikel	conj	Konjunktion
fv	Finites Verb	infz	Infimitiv-Zusatz
nfv	Nicht-finites Verb	nomen	Nomen
NOMI	Nominal	NP	Nominalphrase
npmod	NP-Modifizierung	PP	Präpositionalphrase
pr	Präposition	prart	Präpositionalartikel
prn	Pronomen	prprn	Präpositionalpronomen
SATZ	ganzer Satz	TSFV	Teilsatz m. fin. Verb
TSNFV	Teils. m. nicht f. Verb	vz	Verb-Zusatz

Anhang D - AND/OR-Baum von SYNTAX
(nur erste 4 Ebenen)



Beispiel für einen partiellen Suchbaum von KPSG



Anhang E – [illegible]

Satz-Stichprobe [illegible] Sätzen. Auf der Basis der folgenden Sätze wurden die Regeln d[illegible] ermittelt und das System getestet:

1. Aller Dinge Anfang ist schwer.
2. Sogleich erscheint der größte Mensch.
3. Sein Freund war allerdings dagegen.
4. Er tat es in dessen Namen.
5. Geld macht nicht glücklich.
6. Gegen jene Tätigkeit sprach nichts.
7. Er scheint sich seiner Sache sicher zu sein.
8. Selbst die kleinen Truppen des französischen Königs ziehen in den Kampf.
9. Bestimmt bringt die letzte Schlacht schnell einen Erfolg.
10. Übrigens können diese Kräfte kaum Einfluß auf die Regierung haben.
11. Also zeigt uns der Paragraph die Pflicht auf.
12. Die Schrift darf diese Beziehung jetzt durchaus enthalten.
13. Daraufhin blieb der Anspruch auf den Platz.
14. Deshalb bleiben fünf Prozent der Wechsel auf der Bank.
15. Später waren aus Wünschen Maßregeln geworden.
16. Sie dürfen zunächst gern einige Stunden reden.
17. Du kannst kurz die Art der Gestalt erklären.
18. Der Kopf ist wirklich ganz geblieben.
19. Erst die Dritten vertrauen den Zweiten aufs Neue.
20. Insbesondere als Künstler hat er Ausdruck gehabt.
21. Anders als in der ersten Hälfte folgt er der nächsten Straße.
22. Mit Flügeln hat er sich nach links entfernt.
23. Die Zukunft der Armen ist ernst.