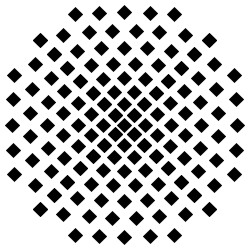

Universität Stuttgart
Fakultät Informatik

Institut für Informatik
Breitwiesenstraße 20-22
D-70565 Stuttgart

ChaPLin 3.2

Ein Chartparser für linguistische Untersuchungen

Gerrit Burkert
Mathis Löthe

Report Nr. 1996/1

26.4.1996

CR-Klassifikation: I.2.7,F.4.3,J.5.3

email: Gerrit.Burkert@swisslife.ch
Mathis.Loethe@informatik.uni-stuttgart.de

Abstract

ChaPLin is a **Cha**rt **P**arser for **Lin**guistic research. Using dynamic programming, chart parsers like ChaPLin store partial results in a *wellformed substring table* (chart). They can use all kind of context free grammars including ambiguous and nondeterministic ones. ChaPLin is specially designed for the needs of natural language processing.

- It can use various grammar formalisms such as feature grammars, that provide extra linguistic information for the parsing process and improve speed and accuracy. Grammar formalisms can be programmed by the user.
- The output generator can be programmed by the user and can be costumized for different kinds of lexical information.
- An ATN-based line scanner is provided.
- ChaPLin has a parse tree visualizer, documentation of data structures, and debugging tools to enhance debugging of grammars and grammar formalisms.

ChaPLin has been developed by Gerrit Burkert at the University of Stuttgart between 1989 and 1994 in Common Lisp. This document describes Version 3.2, which has been completed by Mathis Löthe in February 1995.
In this report, the basic concepts of chart parsing are described and a user manual as well as a reference manual for ChaPLin are provided.

Zusammenfassung

ChaPLin ist ein **Cha**rt-**P**arser für **Lin**guistische Untersuchungen. Chartparser wie ChaPLin legen nach dem Prinzip der dynamischen Programmierung Teilresultate in einer Tabelle, der Chart, ab. ChaPLin arbeitet mit beliebigen kontextfreien Grammatiken (auch mit mehrdeutigen und nichtdeterministischen). Ergebnis des Parseprozesses ist der Syntaxbaum des Eingabesatzes. ChaPLin ist speziell auf die Bedürfnisse der Sprachverarbeitung ausgerichtet.

- Grammatikformalismen steuern den Parser durch zusätzliche linguistische Information. Grammatikformalismen für ChaPLin können vom Benutzer erstellt werden.
- Der Ausgabegenerator kann vom Benutzer programmiert werden, um verschiedene Arten von Lexikoninformation zu verwerten und um den Bedürfnissen der Weiterverarbeitung Rechnung zu tragen.
- Für die Vorverarbeitung wird ein ATN-basierter Textscanner zur Verfügung gestellt.
- Um die Suche nach Fehlern an Grammatikformalismus und Grammatik zu erleichtern, gibt es für ChaPLin eine graphische Anzeige für Syntaxbäume, Dokumentation der Datenstrukturen und Fehlersuchhilfen.

ChaPLin wurde von G. Burkert zwischen 1989 und 1994 an der Universität Stuttgart in Common Lisp entwickelt. Die hier beschriebene Version 3.2 wurde von Mathis Löthe bis zum Februar 1995 erstellt.
Dieser Bericht erläutert die Grundbegriffe des Parsens mit einer Chart und enthält ein Benutzer- und Referenzhandbuch für ChaPLin.

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Überblick

ChaPLin – ein ChartParser für Linguistische Untersuchungen – ist ein Parser für die Verarbeitung natürlicher Sprache. Die Syntaxanalyse (Parsing) ist ein wichtiger Schritt bei der Untersuchung natürlichsprachlicher Texte. Das Ergebnis der Syntaxanalyse ist ein *Syntaxbaum* oder *Parsebaum*.

Die folgenden Eigenschaften des Chartparseverfahrens für kontextfreie Grammatiken sind wichtig, um den speziellen Anforderungen der Verarbeitung natürlicher Sprache und den Bedürfnissen der linguistischen Forschung gerecht zu werden:

- Die Chart ist eine Datenstruktur, die alle Zwischenergebnisse und Zwischenschritte beim Parsen explizit repräsentiert. Wenn kein Syntaxbaum für den ganzen Satz gefunden wird, können Teilergebnisse aus der Chart extrahiert werden. So kann man auch *partiell* parsen, d.h. nach Teilstrukturen wie z.B. Nominalphrasen suchen. Eine Analyse der Zwischenergebnisse hilft außerdem bei der Entwicklung von Grammatiken.

- Natürlichsprachliche Sätze sind syntaktisch mehrdeutig, so daß ein Parser für natürliche Sprache mit mehrdeutigen Grammatiken arbeiten können muß. Bei einem mehrdeutigen Satz gibt es daher nicht einen einzigen Parsebaum sondern eine Menge von Parsebäumen (Lesarten), den sogenannten *Parsewald*. Die Chart ist eine kompakte Darstellung eines Parsewalds, aus der die einzelnen Lesarten nach Bedarf extrahiert werden können.

- Die Grammatik wird nicht vorverarbeitet, so daß der Ableitungsweg durch eine Analyse der Chart leicht nachvollzogen werden kann. ChaPLin kann Grammatiken mit mehreren Startsymbolen und Grammatiken mit ε-Zyklen verwenden, so daß bei der Erstellung einer Grammatik auf deren Eigenschaften keine Rücksicht genommen werden muß.

Zur vollständigen Beschreibung der Syntax natürlicher Sprache sind kontextfreie Grammatiken als Darstellungsformalismus nicht ausreichend. Die Linguistik kennt daher mächtigere Grammatikformalismen, die den Einsatz zusätzlicher linguistischer Information erlauben. In ChaPLin wird diese Information in den *Features* abgelegt, mit welchen man weitere Bedingungen für die Anwendbarkeit einer Regel formuliert. Das Datenformat für die Features und ihre

Behandlungsregeln wird im *Grammatiktyp* festgelegt. Ein Grammatiktyp ist damit ein Modul für ChaPLin, das einen Grammatikformalismus implementiert.

Bei einer endlichen Menge von Featurewerten kann leicht eine äquivalente kontextfreie Grammatik konstruiert werden. Man kann daher die aus der theoretischen Informatik bekannten Resultate für kontextfreie Sprachen weiterhin verwenden.

Der Parsebaum wird im Normalfall in Form einer geschachtelten Liste ausgegeben, wobei das Ausgabeformat für Blätter und Zwischenknoten des Baumes beim Aufruf des Parsers spezifiziert wird. Für diese Knoten des Syntaxbaums kann man im Grammatiktyp Knotenattribute definieren, die bei der Ausgabegenerierung als synthetisierte Attribute (Z-Attribute) berechnet werden (vgl. semantische Aktionen in [Aho et al. 86]). Damit ist es möglich, bei einem Einsatz in einem größeren System semantische Information für spätere Analysephasen bereitzustellen.

Nicht nur bei der Ausgabe, sondern auch bei der Eingabe bietet ChaPLin verschiedene Möglichkeiten an.

- ChaPLin bietet bei Bedarf verschiedene Vorverarbeitungschritte für den Eingabesatz an. Ein ATN-basierter Zeilenscanner zerlegt vom Benutzer eingegebene Sätze in eine Folge von Wortformen und Satzzeichen, den *Eingabeelementen*. Der Parser selbst arbeitet mit einer Folge von solchen Eingabeelementen. Die Wortformen werden vor dem Parsen mit einem Lexikon analysiert.

- Es ist möglich nacheinander mehrere Grammatiken zu verwenden. Nachdem man mit der ersten Grammatik geparst hat, enthält die Chart die Ergebnisse der ersten Analyse. Eine andere Grammatik kann mit diesen Ergebnissen weiterarbeiten.

  So kann man eine einfache, effiziente Grammatik benutzen, die nur die häufigsten Möglichkeiten berücksichtigt und im Falle eines Mißerfolgs eine genauere Analyse mit einer vollständigeren aber auch aufwendigeren Grammatik vornehmen.

  Außerdem gibt es eine Ausgabestruktur für Zwischenergebnisse, die es ermöglicht, Folgen von Nichtterminalsymbolen mit einer anderen Grammatik weiterzuverarbeiten.

- Man kann inkrementell parsen, d.h. eine bestehende Chart um ein Lexem verlängern und alle möglichen Ableitungen bestimmen. Für den Einsatz in einem interaktiven System gibt es einen inkrementellen Scanner, der ein Eingabeelement sofort an den Parser übergibt, nachdem es vollständig eingegeben ist. ChaPLin beginnt dann schon mit der Analyse, während der Benutzer noch die Eingabe vervollständigt.

- Zur Unterstützung der Suche nach Fehlern in Grammatik und Lexikon werden Werkzeuge zur Zeitmessung, zur Visualisierung der Ergebnisse und für Statistiken über den Ableitungsprozeß bereitgestellt.

Diese unterschiedlichen Möglichkeiten machen ChaPLin für unterschiedliche Benutzerkreise interessant. Wenn man ChaPLin als Teil eines größeren Systems benutzt, interessiert man sich in erster Linie für die verschiedenen Arten, den Parser aufzurufen. Bei der Erstellung einer Grammatik oder eines Lexikons sind dagegen die Analysewerkzeuge wichtig. Um Verarbeitungskomponenten für einen Grammatiktyp zu erstellen, ist es zudem noch nötig, die genaue Arbeitsweise von ChaPLin zu kennen.

Die einzelnen Kapitel dieses Berichts versuchen den unterschiedlichen Bedürfnissen dieser Benutzerkreise Rechnung zu tragen.

- In Kapitel 2 wird die dem Parsen mit einer Chart zugrunde liegende Theorie erläutert.
- Die Benutzeranleitung in Kapitel 3 beschreibt die Einsatzmöglichkeiten und die grundlegenden Eigenschaften der wichtigsten Schnittstellenfunktionen und dient als Einführung in ChaPLin. Genauere Information zu den einzelnen Schnittstellenfunktionen steht im Referenzhandbuch in Kapitel 6.
- Kapitel 4 beschreibt die Datenstrukturen, die Arbeitsweise des Parsers und die Schnittstellen zum Grammatiktyp. Hier werden auch die Bestandteile für Grammatiktypen beschrieben, weil man Grammatiktypen nur mit Kenntnissen über die Arbeitsweise von ChaPLin erstellen kann.
- Das nötige Sprachwissen erhält der Parser durch Angabe einer *Grammatik* und eines *Lexikons*. Kapitel 5 erklärt, wie Grammatik und Lexikon definiert werden und beschreibt einen Grammatiktyp, der mit flachen Featuremengen arbeitet.
- Kapitel 7 gibt einen Überblick über die Implementierungsgeschichte und zukünftige Entwicklungsmöglichkeiten für ChaPLin.
- In den Anhängen befinden sich Beispiele für Grammatik und Lexikon, eine Fehlertabelle, ein Verzeichnis aller Dateien und ein Überblick über den Quellcode.
- Die aus der Literatur übernommenden oder in diesem Bericht eingeführten Fachbegriffe werden im Glossar erläutert.

Kapitel 2

Syntaxanalyse mit Hilfe einer Chart

2.1 Grundlagen

Wie anfangs beschrieben legt ChaPLin Zwischenergebnisse in einer Chart ab. Die Chart ist folgendermaßen definiert:

- Eine Chart ist ein gerichteter Pseudograph [Harary 74] und besteht aus einer (linear) geordneten Menge von Knoten und einer Menge von Kanten. Eine Kante verbindet zwei Knoten, den Anfangs- und den Endknoten der Kante. Mehrfache Kanten (Multikanten) sind zulässig, d.h. es darf zwei Kanten geben, die das gleiche Knotenpaar verbinden.
- Die Kantenmenge besteht aus *aktiven*, *inaktiven* und *Lesart*kanten.
- Alle Kanten sind vorwärtsgerichtet, d.h. der Anfangsknoten einer Kante hat eine kleinere Nummer (in der o.g. Ordnung) als ihr Endknoten. Aktive Kanten dürfen dazu noch *Schlingen* bilden, d.h. Anfangs- und Endknoten sind gleich. Abgesehen von diesen aktiven Schlingen ist die Chart zyklenfrei.
- Von einem beliebigen Knoten aus gibt es zu jedem Knoten mit größerer Nummer einen Pfad aus inaktiven Kanten. Die inaktiven Kanten beschreiben damit eine lineare Ordnung auf der Knotenmenge.
- Kanten enthalten weitere Information. Insbesondere haben Kanten einen oder mehrere *Inhalte*. Wenn der Inhalt der Kante eine Folge von inaktiven Kanten ist, dann beschreiben die enthaltenen Kanten einen Pfad vom Anfangsknoten bis zum Endknoten der Kante.

2.1.1 Die Chart als Darstellungsform des Syntaxbaums

Eine Chart kann als Variante eines Syntaxbaums gesehen werden, mit zusätzlichen Eigenschaften zur Darstellung alternativer Teilstrukturen und offener Hypothesen. Die Knoten der Chart entsprechen den Zwischenräumen im Satz.

Die inaktiven Kanten in der Chart stehen für die Knoten des Syntaxbaums, nämlich für die Terminalsymbole (Blätter des Syntaxbaums, Eingabeelemente) und die Nichtterminalsymbole (innere Knoten des Syntaxbaums). Zu Beginn des Parsevorgangs werden die Terminalsymbole als inaktive Kanten eingetragen; ihr Inhalt ist das Eingabeelement. Eine Terminalkante verbindet immer einen Knoten mit dem nächstgrößeren. Die Nichtterminalkanten werden während des Parsevorgangs erzeugt und erhalten die ihren Konstituenten entsprechenden inaktiven Kanten als Inhalt.

Aktive Kanten sind Hypothesen über anwendbare Regeln. Aktive Kanten haben einen Teil der vom Regelrumpf (rechte Seite der Regel) geforderten inaktiven Kanten schon gefunden, den Rest noch nicht.

Der Ablauf der Syntaxanalyse wird im folgenden am Beispielsatz

The old man the boats

erläutert. Das Lexikon enthalte unter anderem folgende Kategorieinformation:

the	DET
old	ADJ oder N
man	N oder V
the	DET
boats	N

Die Grammatik enthalte folgende Regeln:

S	$\longrightarrow$	NP VP
NP	$\longrightarrow$	DET N
NP	$\longrightarrow$	DET ADJ N
NP	$\longrightarrow$	PROPN
VP	$\longrightarrow$	V NP

Die Abbildungen 2.1 und 2.2 zeigen für dieses Beispiel den Syntaxbaum und dessen Darstellung in der Chart. Während der Analyse eines Satzes werden Kanten in die Chart eingebaut, die immer größere Teile des Eingabesatzes überspannen. Zu Beginn des Analysevorgangs bestimmt der Parser zu jedem Wort mit Hilfe des Lexikons die Kategorien (Wortarten) und trägt entsprechende Terminalkanten in die Chart ein. Der Lexikoneintrag – das Eingabeelement – wird als Inhalt der Terminalkante abgelegt, nicht als eigene Kante. Für manche Wörter gibt es mehrere Kategorien, z.B. kann das Wort *old* in unserem Beispiel sowohl als Substantiv als auch als Adjektiv gelesen werden. Solche Mehrdeutigkeiten werden wie in Abbildung 2.3 einfach durch alternative Kanten dargestellt. So entstehen die o.g. Multikanten in der Chart.

Beim Parsen werden die vorhandenen Kanten nach den Regeln der Grammatik zu übergreifenden Kanten zusammengefaßt. Bei einem solchen Ableitungsschritt entsteht durch Anwendung einer Regel eine Nichtterminalkante, deren Kategorie der Kopf der Regel ist. Der Inhalt der neuen Kante sind die Kanten der Symbole des Regelrumpfs. Möchte man diesen Inhalt im Diagramm verdeutlichen, wird wie in Abbildung 2.4 statt der Kategorie der Kante die ganze Regel hingeschrieben.

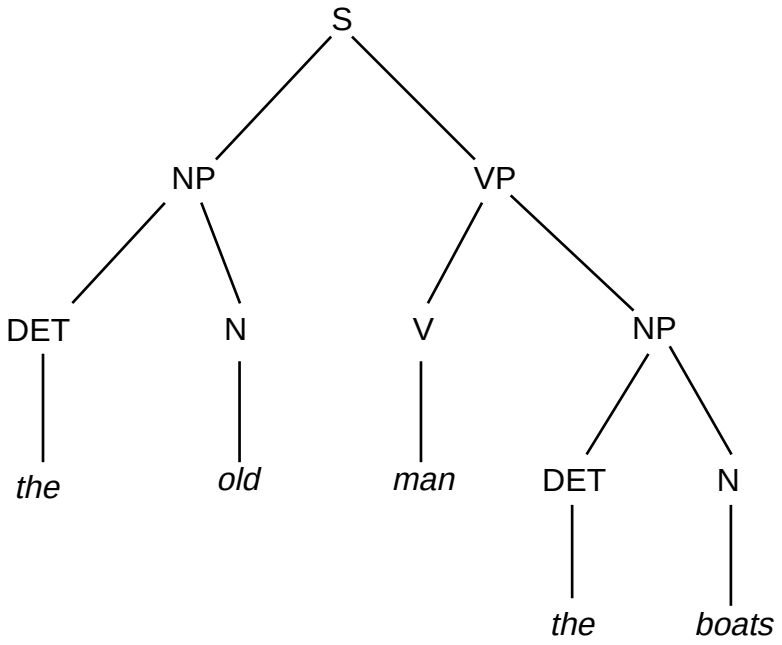

Abbildung 2.1: Syntaxbaum für den Beispielsatz

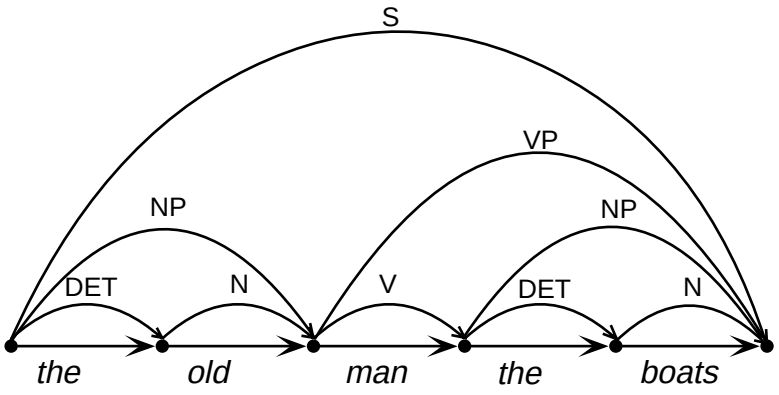

Abbildung 2.2: Chart für den Syntaxbaum aus Abbildung 2.1

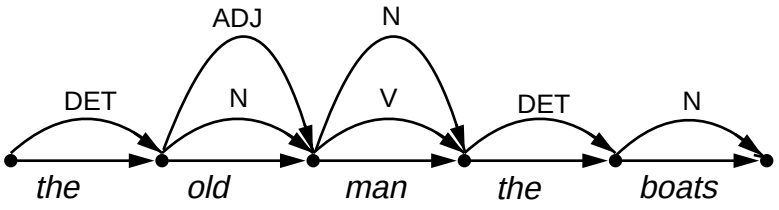

Abbildung 2.3: Chart mit Terminalkanten

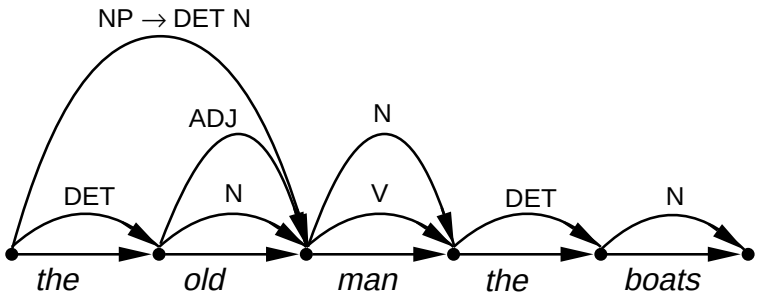

Abbildung 2.4: Chart mit einer Nichtterminalkante

2.1.2 Die Grundregel

Inaktive Kanten stehen für erfolgreich erkannte Symbole. Deswegen ist die in Abbildung 2.4 hergeleitete Nichtterminalkante ebenfalls inaktiv. Die aktiven Kanten enthalten dagegen Informationen über einen aktuellen Versuch, eine Regel anzuwenden. Sie sind eine explizite Darstellung eines Zwischenzustands bei der Abarbeitung einer Regel. Der Notation aus [Aho et al. 86] für Elemente einer Grammatik (parse items) folgend wird die Stelle, bis zu der der Regelrumpf bereits vervollständigt ist, durch einen Punkt markiert.

Grundprinzip des Chartparsens ist, alle unvollständigen Strukturen – nämlich die aktiven Kanten – zu vervollständigen. Immer, wenn das Ende einer aktiven Kante auf den Anfang einer inaktiven Kante trifft und die beiden Kanten zueinander passen, wird eine neue Kante erzeugt. Dies führt zu folgender Grundregel (fundamental rule) des Chartparsings:

Algorithmus 1

wenn der Endknoten einer aktiven Kante A und der Anfangsknoten einer inaktiven Kante I gleich sind (die Kanten sich treffen)

und die Kategorie von I das erste benötigte Symbol von A ist (die Kanten zueinander passen),

dann erzeuge eine neue Kante K,

- deren Anfangsknoten der Anfangsknoten von A ist,
- deren Endknoten der Endknoten von I ist,
- deren Kategorie die Kategorie von A ist und
- deren Inhalt aus dem Inhalt von A und der Kante I besteht.
- Wenn I das letzte von A benötigte Symbol ist,
  dann ist K inaktiv, sonst ist K aktiv.

Trage die Kante K in die Chart ein,

- wenn K aktiv ist,
- wenn K inaktiv ist,
  und keine zu K äquivalente inaktive Kante I' existiert.

Sonst verschmelze die Kante K mit der o.g. äquivalenten Kante I'.

Die genaue Realisierung der Grundregel wird in Abschnitt 4.3 beschrieben. Abbildung 2.5 veranschaulicht die Grundregel anhand des Beispiels, wobei aktive Kanten **fett** gedruckt werden. Die aktive Kante NP $\longrightarrow$. DET N versucht, die Regel NP $\longrightarrow$ DET N anzuwenden, d.h eine Nominalphrase NP zu erkennen, die aus einem Artikel DET und einem Substantiv N besteht. Da die aktive Schlinge noch kein Symbol gelesen hat, steht der Punkt vor dem ersten Symbol im Regelrumpf.

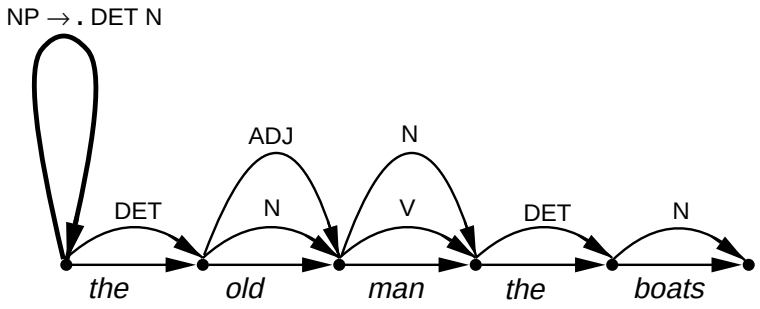

Abbildung 2.5: Chart mit einer aktiven Schlinge

Das Ende dieser aktiven NP-Kante trifft mit der inaktiven DET-Kante zusammen. Damit ist die erste Bedingung der Grundregel für die beiden Kanten erfüllt. Auch paßt die aktive Kante NP $\longrightarrow$. DET N zur inaktiven Kante DET, weil DET als nächstes Symbol nach dem Punkt steht. Abbildung 2.6 zeigt, wie die neue aktive Kante NP $\longrightarrow$ DET . N in die Chart eingefügt wird.

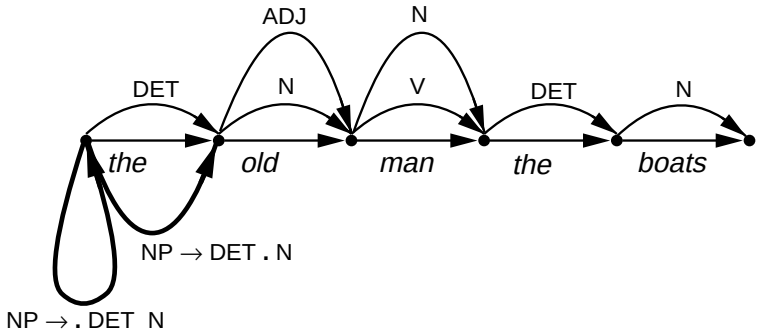

Abbildung 2.6: Chart nach dem ersten Lesevorgang

Dabei ist zu beachten, daß der Chart eine neue Kante hinzugefügt und nicht die erste aktive Kante durch die neue ersetzt wird. Das ist wesentlich, da *alle* möglichen syntaktischen Strukturen eines Satzes gefunden werden sollen. Chartparser arbeiten grundsätzlich monoton, d.h. sie fügen neue Kanten hinzu, ändern oder entfernen aber niemals bestehende Kanten.

Wenn eine Kante hergeleitet wird, die zu einer bereits eingetragenen inaktiven Kante äquivalent ist, dann wird statt einer zusätzlichen Kante nur ein neuer Inhalt – eine weitere Lesart – für die bestehende Kante angelegt. Diese *Kantenverschmelzung* verbessert die Effizienz bei mehrdeutigen Grammatiken. Nach [Seiffert 89] beträgt der Platzbedarf dann $O(n^3)$ wobei n die Anzahl der Ein-

gabeelemente ist. Ohne Kantenverschmelzung wird für jede Lesart des Satzes eine eigene Kante angelegt, wobei die Anzahl der Lesarten in $O(k^n)$ liegt. In der Praxis funktioniert bei Verzicht auf Kantenverschmelzung das Verfahren zwar in den meisten Fällen gut, jedoch überschreitet für einzelne Sätze der Speicherbedarf die Leistungsfähigkeit des Rechners.

In Abbildung 2.7 werden die Kantenpaare NP-ADJ und NP-N betrachtet. Beim ersten Paar passen die Kanten nicht, während beim zweiten nach der Grundregel eine inaktive NP-Kante erzeugt wird.

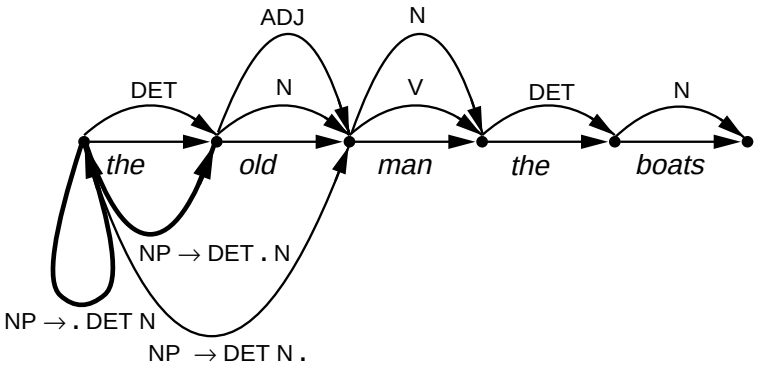

Abbildung 2.7: Chart nach der ersten Ableitung

Jedes Einfügen einer Kante – aktiv oder inaktiv – kann nach der Grundregel weitere Regelanwendungen ermöglichen und dadurch den Einbau weiterer Kanten nach sich ziehen.

2.2 Parseprozeß

Beim Parsen werden Symbole gelesen, d.h. aktive Kanten werden um eine inaktive Kante verlängert. Solche nach der Grundregel passenden Kombinationen von aktiven und inaktiven Kanten nennt man *Konfigurationen*. Jedesmal, wenn eine Kante eingefügt worden ist, werden die dadurch neu entstandenen Konfigurationen in der *Agenda* abgelegt und nacheinander untersucht. Ist die Agenda leer, d.h. keine nicht-untersuchte Konfiguration mehr vorhanden, ist der Parseprozess beendet, da keine weiteren Ableitungen mehr möglich sind.

Die Agenda ist bei ChaPLin als Stapel organisiert, d.h. die zuletzt entdeckten Konfigurationen werden zuerst bearbeitet. Im Suchbaum der Ableitungen entspricht das der Tiefensuche. Die Grundregel verlangt aber keine besondere Reihenfolge der Abarbeitung, so daß auch eine Breitensuche mit einer Warteschlange als Agenda oder verschiedene Heuristiken mit einer Prioritätswarteschlange als Agenda denkbar wären.

Chartparser können eine Eingabesequenz sowohl top-down als auch bottom-up bearbeiten, wobei die beiden Modi dasselbe Endergebnis liefern. Der Parseprozeß kommt nur in Gang, wenn die Chart aktiviert wird, d.h. für jede Grammatikregel, die anwendbar wird, eine aktive Schlinge eingetragen wird. Die beiden Modi prüfen die Anwendbarkeit einer Regel auf unterschiedliche Art und zu unterschiedlichen Zeitpunkten

2.2.1 Top-down Analyse

Beim top-down Parsen geht die Analyse vom *Ziel* aus; dazu gibt man dem Parser die gesuchte Kategorie als Startsymbol vor. Für jede Regel, die diese Kategorie herleitet (die Kategorie als Regelkopf hat), wird eine aktive Schlinge angelegt. Aktive Kanten werden entweder durch die Grundregel oder bei der Aktivierung in die Chart eingetragen. Nach dem Eintragen einer aktiven Kante wird das erste von ihr noch benötigte Symbol – das rechts vom Punkt – zum neuen Zwischenziel. Wie zu Beginn beim Startsymbol legt der Parser für jede Regel, die dieses Zwischenziel herleitet, eine aktive Schlinge an. Im top-down Modus wird die Chart also zu Beginn des Parsevorgangs und nach jedem Einfügen einer aktiven Kante aktiviert.

Der Analysevorgang beginnt in Abbildung 2.8 mit einer aktiven S-Kante am ersten Knoten für jede Regel der Form S $\longrightarrow$ .... Zunächst wird die aktive S-Kante für die Regel S $\longrightarrow$ NP VP angelegt.

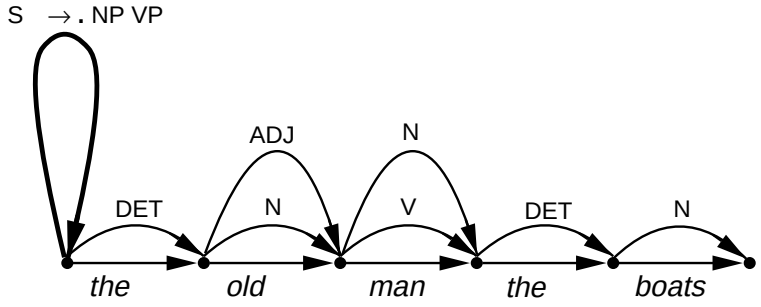

Abbildung 2.8: Beginn der top-down Aktivierung

Das erste Symbol der rechten Seite der Regel, NP, wird zum Zwischenziel. Damit der Parser NP herleiten kann, sucht er nach Regeln mit Kopf NP in der Grammatik und baut, wie in Abbildung 2.9 gezeigt, drei neue Kanten ein.

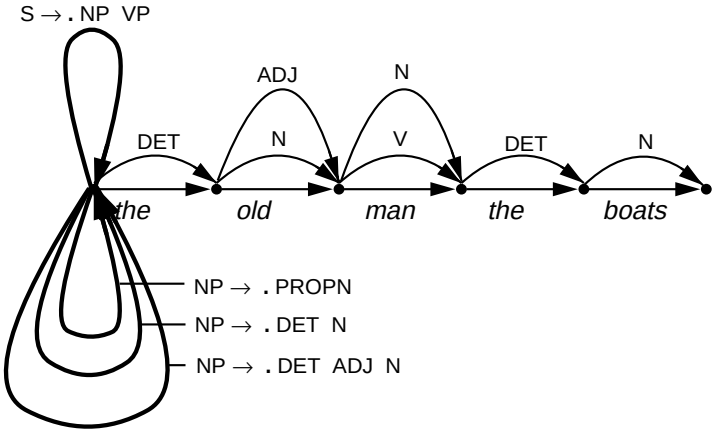

Abbildung 2.9: Top-down aktivierte Chart

Im weiteren Analyseprozeß wird nach der Grundregel die aktive Kante NP $\longrightarrow$. DET N mit der inaktiven Kante DET zur aktiven Kante NP $\longrightarrow$ DET . N erweitert. Das neue Zwischenziel ist jetzt das Symbol N, das rechts vom Punkt steht. Nach dem Einfügen dieser aktiven Kante durch die Grundregel wird nach Grammatikregeln gesucht die das Symbol N herleiten.

2.2.2 Bottom-up Analyse

Das bottom-up Parsen geht von den vorhandenen Symbolen aus und versucht, daraus neue Nichtterminalsymbole herzuleiten, bis am Ende möglicherweise das Startsymbol hergeleitet wird. Nachdem der Parser eine inaktive Kante eingefügt hat, untersucht er, ob sich dadurch neue Möglichkeiten ergeben, Regeln anzuwenden. Dies ist der Fall, wenn eine Grammatikregel existiert, deren erstes Symbol im Regelrumpf mit der Kategorie der inaktiven Kante übereinstimmt. Für jede solche Grammatikregel wird am Ausgangsknoten der inaktiven Kante eine aktive Schlinge erzeugt (sofern sie nicht schon existiert). Im bottom-up Modus wird also nach dem Einfügen einer *inaktiven* Kante aktiviert. Da die Terminalsymbole inaktive Kanten sind, wird die Chart automatisch aktiviert, wenn die Kategoriekanten eingetragen werden.

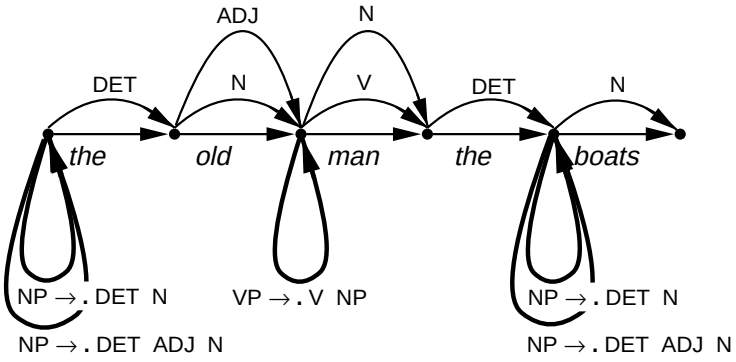

Abbildung 2.10: Bottom-up aktivierte Chart

Abbildung 2.10 zeigt die aktivierte Chart für den Beispielsatz. Beim Einfügen der inaktiven DET-Kante werden für die beiden Regeln, die die Kategorie NP herleiten, aktive NP-Kanten erzeugt, welche anschließend nach der Grundregel erweitert werden können.

Der Parseprozeß beim bottom-up Parsen endet, wenn keine weiteren Schritte mehr ausführbar sind. Ist dann eine inaktive Kante entstanden, die die ganze Chart überspannt und deren Kategorie das (ein) Startsymbol ist, so war das Parsen erfolgreich.

Im Gegensatz zum top-down Parsen werden beim bottom-up-Parsen auch alle Nichtterminalsymbole für *Teile* des Satzes abgeleitet. Wenn also keine Satzstruktur erkannt werden konnte, ist beim bottom-up-Parsen wenigstens eine Folge der erkennbaren Teilstrukturen (Fragmente) verfügbar (partielles Parsing).

Kapitel 3

Bedienungsanleitung

ChaPLin kann auf vielfältige Art und Weise eingesetzt werden. Der Parser bietet Schnittstellenfunktionen für die verschiedenen Analysephasen an, die man durch mehrere Parameter beeinflussen kann. Dieses Kapitel stellt den typischen Verwendungszweck der wichtigsten Funktionen und ihrer wesentlichen Parameter vor. Eine genauere Beschreibung findet sich dann im Referenzhandbuch in Kapitel 6. Einige kompliziertere Optionen des Ausgabegenerators können nur mit Kenntnissen aus den Kapiteln 4 und 5 verwendet werden und werden deswegen ebenfalls erst in Kapitel 6 dokumentiert.

3.1 Analysephasen

ChaPLin analysiert einen Satz in mehreren Phasen. Die Schnittstellenfunktionen des Parsers stellen dem Benutzer unterschiedliche Kombinationen der Analysephasen zur Verfügung. Abbildung 3.1 zeigt die Analysephasen und den groben Aufbau von ChaPLin.

Algorithmus 2

1. **Scanning:** In Textdateien oder Benutzereingaben liegt ein Text als Folge von Zeichen vor. Die Aufgabe des Scanners ist, einen als String gegebenen Satz in eine Folge von Eingabeelementen (Wortformen und Satzzeichen) zu zerlegen. ChaPLin stellt dafür einen ATN-basierten Zeilenscanner zur Verfügung. Dieser Scanner muß allerdings nicht benutzt werden. Man kann ChaPLin auch mit einer gescannten Eingabesequenz aufrufen.

2. **Belegen der Chart:** Zu Beginn werden die Kategoriekanten als Terminalsymbole in die Chart eingetragen. Die lexikalische Analyse bestimmt dazu für jede Wortform die Kategorie und eine Reihe weiterer Informationen wie morpho-syntaktische Merkmale und eine semantische Spezifikation. Man kann ChaPLin auch mit einer Sequenz von Lexikoneinträgen aufrufen.

3. **syntaktische Analyse:** ChaPLin analysiert die Eingabesequenz anhand einer Grammatik eines bestimmten Grammatiktyps.

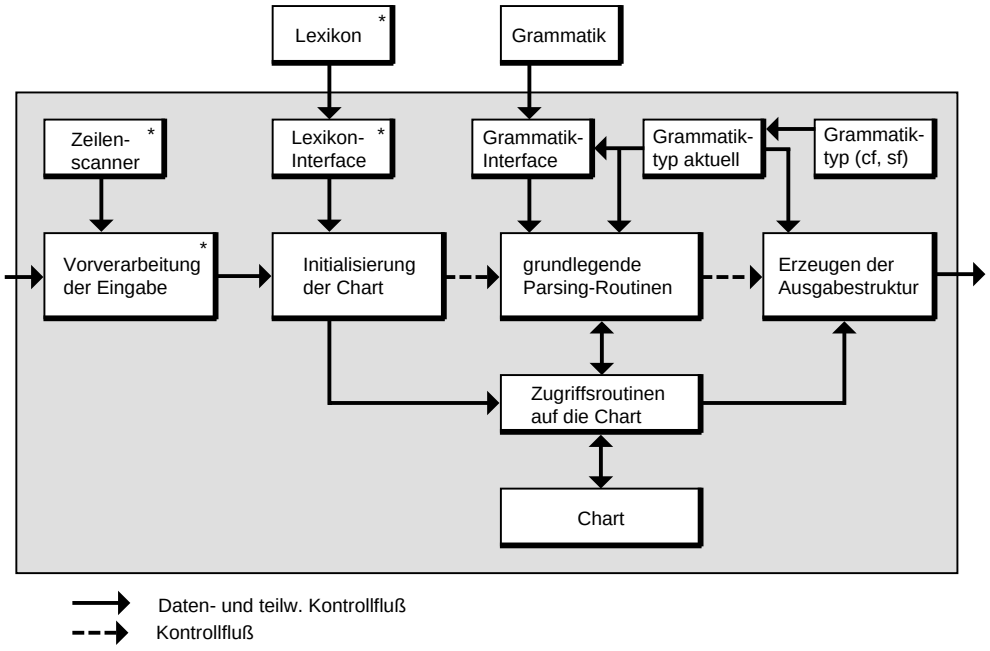

Abbildung 3.1: Aufbau von ChaPLin
*Mit * gekennzeichnete Komponenten werden nicht benutzt, wenn ChaPLin als Teil eines Analysesystems mit getrennter lexikalischer Analyse eingesetzt wird.*

4. **Ausgabegenerierung** Die Flexibilität der Ausgabegenerierung ist besonders wichtig im Hinblick auf die angestrebte Verwendung. Eine einfache Erfolgsmeldung oder die Anzahl der Lesarten läßt sich mit wenig Aufwand berechnen. Aufwendiger ist es dagegen, den Parsewald auszugeben, der in zwei Stufen berechnet wird:

   Attributberechnung Nach der Analyse können zusätzliche Attribute für die einzelnen Ableitungsschritte berechnet werden. Im Gegensatz zu den Features steuern diese Attribute nicht die syntaktische Analyse. Die in ihnen enthaltene Information wird nur zur Ausgabegenerierung benutzt. Die Ausgabe ist Information für den Benutzer oder Grundlage für weitere Analyseschritte.

   Wird das gleiche Symbol auf verschiedenen Ableitungswegen erreicht, dann können sich verschiedene Attributwerte ergeben. Daher erzeugt ChaPLin zur Attributberechnung für jeden Ableitungsweg eine Lesartkante, die außer der syntaktischen Information noch die Attribute enthält. Damit wird die Kantenverschmelzung rückgängig gemacht.

   Baumerzeugung Der Parser wählt bei der Baumerzeugung die gewünschte Information aus der Lesartkante aus. Außerdem kann spezifiziert werden, daß bei der Baumerzeugung anstelle bestimmter Kanten nur ihr Inhalt eingefügt wird (*Ignore*kategorien).

Die Schnittstellenfunktionen des Parsers sind sinnvolle Kombinationen der einzelnen Phasen und man kann je nach Bedarf die entsprechenden Funktionen verwenden.

3.2 Aufruf des Parsers

Dieser Abschnitt beschreibt, wie die verschiedenen Schnittstellenfunktionen und Parameter des Parsers verwendet werden. Die Keywordparameter werden erst in Kapitel 6 ausführlich beschrieben, es sind aber jeweils sinnvolle Defaultwerte definiert. Die Beispiele in diesem Kapitel basieren auf dem Lexikon und der Grammatik aus Anhang A.

3.2.1 Verarbeitung unanalysierter Einzelsätze

Für Testzwecke ist es am einfachsten, unvorverarbeitete Sätze zu analysieren. Der Parser erhält einen String, zerlegt ihn mit seinem ATN-Scanner in Eingabeelemente und parst diese Eingabesequenz. Diese Variante wird bei der Entwicklung eingesetzt, um Parser, Grammatik oder Lexikon zu testen, weil der Entwickler Sätze bequem eingeben kann.

Die Funktion `parse-line` liest nach dem Prompt `->` einen Satz vom Terminal und parst ihn.

```
USER(11): (parse-line)
-> Der Berg ruft.
((s (np (det "Der") (n "Berg")) (vp (vf "ruft")) (punkt ".")))
```

Die Funktion entdeckt eine Lesart für den Satz und gibt deren Syntaxbaum in einer Liste zurück. `parse-line` deckt damit alle Analysephasen aus Abbildung 3.1 ab.

Die folgenden zwei Funktionen scannen ihre Eingabe nur, d.h. sie geben eine Liste von Eingabeelementen zurück. Die Funktion `scan` zerlegt ihr Argument, einen String, in Eingabeelemente, während `scan-line` den String vorher vom Terminal einliest.

```
USER(12): (scan-line)
-> Der Berg ruft.
("Der" "Berg" "ruft" #\.)
USER(13): (scan "Der Berg ruft.")
("Der" "Berg" "ruft" #\.)
```

Diese Eingabesequenz kann dann an eine der im folgenden beschriebenen Funktionen weitergegeben werden.

3.2.2 Parsen von Eingabesequenzen

Bei der Verarbeitung eines fortlaufenden Texts kann es je nach dessen Herkunft schwierig sein, ihn in einzelne Sätze zu zerlegen. Der Scanner muß z.B. zwischen einem Abkürzungspunkt und dem Satzende unterscheiden und wörtliche

Rede über mehrere Sätze bearbeiten können. In diesem Fall wird der Text in einer separaten Phase gescannt und der Parser erhält jeden Satz als Liste von Eingabeelementen.

```
USER(46): (parse '("Der" "Berg" "ruft" #\.))
((s (np (det "Der") (n "Berg")) (vp (vf "ruft")) (punkt ".")))
```

Mit den Defaulteinstellungen arbeitet der Parser im bottom-up-Modus und erzeugt Syntaxbäume für alle Kanten, die die ganze Chart überdecken. Für Nichtterminalknoten gibt er die Kategorie und für Terminalknoten zusätzlich das Eingabeelement aus. Im Beispiel wird nur eine Lesart gefunden, so daß die Ergebnisliste nur einen Parsebaum enthält.
Auf diese Weise kann man auch eine einzelne Nominalphrase analysieren, wobei es mit der im Test verwendeten Grammatik wiederum eine Lesart gibt.

```
(USER(52): (parse '("Der" "Berg"))
((np (det "Der") (n "Berg")))
```

Im folgenden Fall wird kein Ergebnis gefunden, denn der Punkt am Satzende fehlt. Unsere Beispielgrammatik (siehe Anhang A.2) verlangt jedoch einen Punkt als Satzendzeichen.

```
USER(53): (parse  '("Der" "Berg" "ruft"))
nil
```

Wenn man die lexikalische Analyse bereits durchgeführt hat, gibt man eine Folge von Lexikoneinträgen anstelle von Eingabeelementen an. Die Funktion `parse` erkennt den Unterschied anhand eines Schlüsselworts für Lexikoneinträge. Dieser Modus erlaubt es, zur Fehlersuche die Ausgangsdaten genau zu überprüfen und erspart bei mehrmaligem Parsen die mehrfache lexikalische Analyse.

3.2.3 Ausgabegenerierung

Bei den Parserfunktionen, die die Ausgabegenerierungsphase enthalten, kann man die Ausgabespezifikation mit dem Keywordargument `:output` angeben. Sie bestimmt, wie die Ausgabe des Parsers aussieht. Die kürzesten Ausgabespezikationen sind Keywords. Bei der Ausgabeform `:succ` gibt der Parser `t` zurück, wenn der Satz geparst werden konnte, sonst `nil`. Bei der Spezifikation `:count` wird die Anzahl der Lesarten bestimmt, während bei `:cat` eine Liste der Kategorien der Kanten, die die ganze Chart überspannen, zurückgegeben wird. ChaPLin kann die Ausgabe durch Aufruf von `build-tree` beliebig oft neu generieren, ohne daß die Syntaxanalyse wiederholt werden muß.

Defaulteinstellung ist die Ausgabe des Parsewalds als Liste aller Syntaxbäume (`:tree`). ChaPLin erzeugt den Parsewald in zwei Schritten. Zuerst expandiert der Parser die Lesarten, d.h. er macht die Kantenverschmelzung rückgängig, indem er für jeden Ableitungsweg eine Lesartkante in die Chart einträgt. Das ist nötig, um für die Ableitungswege nach den Angaben im Grammatikformalismus und in den einzelnen Regeln weitere Attribute für die Ausgabe zu berechnen.

Bei mehrdeutigen Grammatiken wächst die Anzahl der Lesarten im schlimmsten Fall exponentiell mit der Satzlänge und es besteht die Gefahr, daß ChaPLin für ihre explizite Erzeugung sehr viel Zeit und Speicherplatz benötigt oder sogar aus Speichermangel abbricht. Der Parser erzeugt die Lesartkanten beim Zählen der Lesarten mit `:count` und für die Ausgabeformen `:succ` und `:cat` nicht, so daß dabei nur ein geringer Zeitbedarf und kein Speicherplatzbedarf entsteht. Wenn man eine hohe Zahl von Lesarten befürchtet, empfiehlt es sich daher, die Lesarten mit `:output :count` zu zählen und bei großen Lesartzahlen den Parsewald nicht zu erzeugen.

```
<USER.12> (parse '("Der" "Berg" "ruft" #\.) :output :count)
1
<USER.13> (build-tree)
((s (np (det "Der") (n "Berg")) (vp (vf "ruft")) (punkt ".")))
```

Aus den Lesartkanten erzeugt ChaPLin den Parsewald entprechend der Baumspezifikation. Der Syntaxbaum ist eine geschachtelte Liste aus dem Knoten und den Unterbäumen. Eine Baumspezifikation gibt die Gestalt von Nichtterminal- und Terminalknoten (Blättern) an. Das folgende Beispiel zeigt die Defaulteinstellung für den Syntaxbaum.

```
<USER.7> chp::*tree-default*
(:node chp::edge-cat
 :lex (:cat :lex)
 :struct (:cat . :contents))
```

Hinter dem Keyword `:node` steht die Spezifikation für die Nichtterminalknoten – in unserem Fall die Kategorie der Lesartkante. Die Spezifikation für Blätter bezeichnet man mit dem Keyword `:lex`, denn hier werden die Angaben dem Lexikoneintrag entnommen (Einzelheiten siehe Kapitel 4.4.5). Bei der Defaulteinstellung wird eine Liste aus Kategorie und Eingabeelement als Blatt verwendet.

3.2.4 Inkrementelles Parsen

Die bisher vorgestellten Funktionen verarbeiten ganze Sätze. Beim inkrementellen Parsen erhält ChaPLin dagegen den Satz Stück für Stück und bearbeitet ihn im bottom-up-Modus, soweit es möglich ist. Setzt man ChaPLin in einem interaktiven System ein, beginnen der inkrementelle Parser und die möglicherweise zeitraubende lexikalische Analyse schon mit der Arbeit, während der Benutzer noch die Eingabe vervollständigt. Das Antwortzeitverhalten wird dadurch verbessert.

Die Funktion `parse-next` arbeitet *wortweise* inkrementell. Sie verlängert die Chart um das Eingabeelement, parst und gibt in der Defaulteinstellung eine Liste der Kategorien der Kanten zurück, die den letzten Knoten als Endknoten haben. In einem interaktiven System wünscht man für einen unvollständigen Satz wie im folgenden Beispiel üblicherweise keine Ausgabestruktur sondern nur eine Erfolgsmeldung. Bei der Ausgabespezifikation `:succ` gibt ChaPLin

`t` zurück, wenn es eine Kante gibt, die die ganze Chart überspannt. Möchte man nur ganze Sätze erkennen, gibt man noch mit dem Parameter `:find` das entsprechende Grammatiksymbol als Startsymbol an. Dann werden nur Kanten berücksichtigt, die die ganze Chart überspannen und die richtige Kategorie haben.

```
<USER.36> (build-chart ())
t
<USER.37> (parse-next "Der" :output :succ :find 's)
nil
<USER.38> (parse-next "Berg" :output :succ :find 's)
nil
<USER.39> (parse-next "ruft" :output :succ :find 's)
nil
<USER.40> (parse-next #\. :output :succ :find 's)
t
<USER.41> (build-tree)
((s (np (det "Der") (n "Berg")) (vp (vf "ruft")) (punkt ".")))
```

Der Aufruf von `build-chart` mit einer leeren Eingabesequenz erzeugt eine leere Chart, die dann Eingabeelement für Eingabeelement verlängert wird. Der letzte Schritt `build-tree` ruft den Ausgabegenerator mit der Defaulteinstellung für den Syntaxbaum. `parse-next` kann auch eine Chart, die von einer anderen Funktion wie `parse` oder `parse-line` erzeugt wurde, um ein Eingabeelement verlängern, z.B. wenn man einmal einen Punkt am Satzende vergessen hat.

Der *zeichenweise* inkrementelle Parser liest den Eingabestring direkt vom Terminal. Ein interaktiver Scanner gibt ein Eingabeelement dann an `parse-next` weiter, wenn der Benutzer es vollständig eingegeben hat. Der zeichenweise inkrementelle Parser verwendet also den wortweise inkrementellen Parser. Der Benutzer kann auch Eingabezeichen löschen. Wird ein bereits weitergegebenes Eingabeelement gelöscht, parst `parse-incremental` den ganzen bisher eingegebenen Satz erneut. Dafür wird ein spezieller Scanner benötigt, der nicht mit dem ATN-Scanner aus `scan-line` äquivalent ist. Zur Interaktion mit dem Terminal werden systemabhängige Funktionen benötigt. Eine Benutzerunterbrechung oder ein Absturz im inkrementellen Scanner können das Terminal in einen undefinierten Zustand bringen.

```
<USER.42> (parse-incremental)
=> Der Berg ru
```

Die Wörter „Der“ und „Berg“ des Beispielsatzes wurden bereits an `parse-next` übergeben. Die Eingabe wird bei `parse-incremental` mit Return beendet.

3.2.5 Analysephasen

Wendet man die Phasen einzeln an, eröffnen sich weitere Einsatzmöglichkeiten. Alle bisher genannten Parserfunktionen sind beliebig kombinierbar, sofern die

von der jeweiligen Funktion vorausgesetzten Analysephasen bereits durchgeführt sind.

Die Funktion `build-chart` baut nur die Chart auf und führt ggf. die lexikalische Analyse durch. Sie erhält wie `parse` eine Folge vom Eingabeelementen als Argument und trägt den Satz in die Chart ein. Grammatik und Modus werden benötigt, weil im bottom-up-Modus die Chart bereits beim Eintragen der Terminalsymbole aktiviert wird. Die Funktion `parse-rest` führt dann den eigentlichen Parsevorgang aus und erzeugt die Ausgabe. `parse-rest` erhält keine Eingabesequenz als Element sondern geht von einer bestehenden Chart aus.

Mit dieser Kombination ist es möglich, einen Satz stufenweise mit verschiedenen Grammatiken zu parsen. Da `parse-rest` von den Kanten der Chart ausgeht, verwendet es die in der Chart eingetragenen Teilergebnisse der vorhergehenden Schritte mit. Für die beiden Grammatiken `*g1*` und `*g2*` sieht der Aufruf dann folgendermaßen aus.

```
> (build-chart '(...))
t
> (parse-rest :grammar *g1*)
...
> (parse-rest :grammar *g2*)
...
```

Dabei sind verschiedene Anwendungen denkbar. `*g1*` ist z.B. eine Grammatik mit Regeln für kleine Konstrukte bis hin zu Nominalphrasen, während `*g2*` dann aus den mit `*g1*` abgeleiteten Symbolen ganze Sätze konstruiert.

Ein anderes Modell ist, zuerst mit einer einfachen Grammatik `*g1*` zu arbeiten und im Falle eines Mißerfolgs mit einer um zusätzliche Regeln erweiterten Grammatik `*g2*` weitere Ableitungen zu bestimmen.

ChaPLin kann auch Folgen von Nichtterminalsymbolen parsen. Diese werden als *Struktureinträge* (ähnlich Lexikoneinträgen) übergeben und genau wie normale Lexikoneinträge in der Eingabesequenz behandelt. Der Struktureintrag enthält den Syntaxbaum für den Aufbau des Nichtterminalsymbols. Der Ausgabegenerator arbeitet bei Struktureinträgen nach dem `:struct`-Teil der Baumspezifikation, der normalerweise so definiert ist, daß der von einem Struktureintrag erzeugte Knoten wie ein gewöhnlicher Nichtterminalknoten des Syntaxbaums aussieht.

3.3 Fehlersuche und Analyse

Wenn man mit Hilfe von ChaPLin eine Grammatik oder einen Grammatiktyp entwickelt, benötigt man Werkzeuge zur Untersuchung des Parseprozesses. Bei Effizienzproblemen oder wenn man den Fehler erst lokalisieren muß, sind quantitative Untersuchungen und Statistiken hilfreich. Quantitative Analysen geben nur einen Hinweis oder eine Tendenz an, weil die Auswirkungen einzelner Effekte von anderen Effekten überlagert werden. Einen bestimmten Ableitungsschritt

untersucht man, indem man sich die Chart oder Teile davon ausgeben läßt. Für noch gründlichere Untersuchungen sind die Datenstrukturen der Chart in Kapitel 4 dokumentiert.

Die Beispieluntersuchungen in diesem Abschnitt werden – wenn nicht anders angegeben – im Anschluß an folgenden Aufruf von ChaPLin ausgeführt.

```
<USER.44>(parse '("Der" "Berg" "ruft" #\.))
((s (np (det "Der") (n "Berg")) (vp (vf "ruft")) (punkt ".")))
```

3.3.1 Quantitative Analyse

Die Funktion `describe-chart` gibt die maximale Knotennummer und die maximale Kantennummer in der Chart zurück.

```
<ISAAC.USER.45> (describe-chart)
((:nodes . 5) (:edges . 23))
```

Bei einem Satz mit 4 Eingabeelementen hat die Chart 5 Knoten. Bei diesem Analyseprozeß wurden 24 Kanten erzeugt. (Die Kantenumerierung beginnt mit 0, die Knotennumerierung mit `*left-vertex*` = 1.)

Die Funktion `chart-analysis` erstellt eine ausführliche Statistik. Die Kantenzahlen werden nach aktiven Kanten, (syntaktischen) inaktiven Kanten und Lesartkanten getrennt aufgelistet. Stillgelegte Kanten entstehen, wenn man den Ausgabegenerator mehrmals aufruft, weil ChaPLin dann die Lesartkanten neu berechnet und die alten Lesartkanten aus der Chart entfernt.

```
<USER.86> (chart-analysis)
Chartanalyse

Knoten: 5   Kanten: 24 davon
 aktiv: 10 inaktiv: 7 Lesartkanten: 7 stillgelegt: 0

1 ueberspannende Kante mit zusammen 1 Lesarten
Vorkommen der inaktiven Kanten in contents
1-fach kommen 7 Kanten vor
Benutzt: 7 unbenutzt: 0 inaktive Kanten

Alternativen:
1 Alternativen bei 7 Kanten

Verteilung der Regellaengen
Laenge 1 wurde 1 mal angewendet
Laenge 2 wurde 1 mal angewendet
Laenge 3 wurde 1 mal angewendet
Terminale: 4

allg. Multikanten
1-fach parallel: 5
```

```
2-fach parallel: 1
Zusammenfassung ergaebe 6 Kanten

kategorieaequivalente Kanten
1-fach parallel: 7
Zusammenfassung ergaebe 7 Kanten
```

Die Statistiken über Verwendung von Kanten, Alternativen und Multikanten berücksichtigen nur die inaktiven Kanten. Eine n-fach parallele Multikante bedeutet, daß n inaktive Kanten dasselbe Knotenpaar verbinden. Multikanten deuten auf Mehrdeutigkeiten in der Grammatik hin. Die Analyse für kategorieäquivalente Kanten zeigt, wie viele parallele Kanten mit gleicher Kategorie es gibt. Da Kantenverschmelzung durchgeführt wird, haben parallele Kanten mit gleicher Kategorie unterschiedliche Features. Die Analysen können auch einzeln durchgeführt werden – eine ausführliche Beschreibung befindet sich in Kapitel 6.

Oft ist auch der Zeitbedarf eines Funktionsaufrufs interessant. ChaPLin stellt zwei Makros zur Zeitmessung zur Verfügung. `with-time` stoppt die Zeit für die Evaluierung der Ausdrücke in seinem Rumpf und gibt Zeit und Wert des letzten Ausdrucks zurück. Bei kleinen Laufzeiten erhält man aussagekräftigere Werte, wenn man die Zeit für eine k-fache Auswertung mit `k-with-time` mißt.

```
<USER.71> (with-time (parse '("Der" "Berg" "ruft" #\.)))
 CPU-Zeitbedarf   0.000 sec
0.0
((s (np (det "Der") (n "Berg")) (vp (vf "ruft")) (punkt ".")))
<USER.74> (k-with-time 1000 (parse '("Der" "Berg" "ruft" #\.)))
 CPU-Zeitbedarf   3.770 sec
3.77
```

Es ist auch interessant, den Zeitbedarf der Analysephasen einzeln zu messen.

3.3.2 Datenausgabe

Eine genauere Vorstellung vom Ableitungsprozeß erhält man, wenn man die Chart betrachtet. Die Funktion `display-chart` gibt die Chart bzw. Teile davon aus.

```
<USER.53> (display-chart :edges :all)
[inactive-edge 16: 1--s-->5  ]
[inactive-edge 12: 1--np-->3  ]
[inactive-edge 0: 1--det-->2  ]
[active-edge 15: 1--s-->4 needed: (punkt)  ]
[active-edge 14: 1--s-->3 needed: (vp punkt)  ]
[active-edge 13: 1--s-->1 needed: (np vp punkt)  ]
[active-edge 11: 1--np-->2 needed: (n)  ]
[active-edge 10: 1--np-->2 needed: (adj n)  ]
```

```
[active-edge 2: 1--np-->1 needed: (det adj n)  ]
[active-edge 1: 1--np-->1 needed: (det n)  ]
[sf-edge 23: 1--s-->5  ]
[sf-edge 19: 1--np-->3  ]
[sf-edge 17: 1--det-->2  ]

[inactive-edge 3: 2--n-->3  ]
[sf-edge 18: 2--n-->3  ]

[inactive-edge 9: 3--vp-->4  ]
[inactive-edge 4: 3--vf-->4  ]
[active-edge 8: 3--vp-->4 needed: (np)  ]
[active-edge 6: 3--vp-->3 needed: (vf np)  ]
[active-edge 5: 3--vp-->3 needed: (vf)  ]
[sf-edge 21: 3--vp-->4  ]
[sf-edge 20: 3--vf-->4  ]

[inactive-edge 7: 4--punkt-->5  ]
[sf-edge 22: 4--punkt-->5  ]
nil
```

Am Anfang der Zeile steht Kantentyp und Kantennummer. Die Kantentypen `inactive-edge` und `active-edge` stehen für inaktive und aktive Kanten; alle anderen Kantentypen (wie `sf-edge`) bezeichnen die Lesartkanten des verwendeten Grammatiktyps.

Anschließend folgt die Nummer des Anfangsknotens, die Kategorie der Kante und die Nummer des Endknotens. Mit dem Keywordparameter `:edges` gibt man an, welcher Kantentyp angezeigt wird. Außer der Defaulteinstellung `:inactive` gibt es noch `:active`, `:tree` für die Lesartkanten und `:all`.

Die Funktion `get-edge` gibt die Kante mit der angegebenen Nummer zurück. Die `print-function` der Kante druckt aus Gründen der Übersichtlichkeit nur Kurzinformation in einer Zeile. Ausführlichere Information erhält man mit `display-edge` :

```
<ISAAC.USER.58> (display-edge (get-edge 16))
[inactive-edge 16: 1--s-->5
  :features nil
  :contents #S(chp::rule :lhs s :rhs (np vp punkt)
                         :conf 1 :rhs-name (np vp punkt)
                         :rhs-cond (nil nil nil)
                         :result nil :sem nil)
     [inactive-edge 12: 1--np-->3  ]
     [inactive-edge 9: 3--vp-->4  ]
     [inactive-edge 7: 4--punkt-->5  ]
nil
```

Die Funktion `display-agenda` zeigt die Agenda. Nach erfolgreicher Beendigung der Parsens ist die Agenda leer.

Das folgende Beispiel zeigt daher die Agenda zu Beginn, nachdem die Chart aufgebaut und bottom-up-aktiviert ist.

```
<ISAAC.USER.54> (build-chart '("Der" "Berg" "ruft" #\.))
t
<ISAAC.USER.55> (display-agenda)
Bottom of stack
1 --- np 1 --- 1 --- det 0 --- 2
1 --- np 2 --- 1 --- det 0 --- 2
3 --- vp 5 --- 3 --- vf 4 --- 4
3 --- vp 6 --- 3 --- vf 4 --- 4
nil
```

Die Agenda ist ein Stapel von Konfigurationen, das sind Paare aus einer aktiven Kante und einer inaktiven Kante, die das Ende der aktiven Kante verlängert (vgl. Kapitel 2). Jede Zeile entspricht einer Konfiguration. Links steht der Anfangsknoten der aktiven Kante, danach deren Kategorie und Kantennummer. In der Mitte steht der Endknoten der aktiven Kante, der gleichzeitig der Anfangsknoten der inaktiven Kante ist. Für die inaktive Kante folgt ebenfalls Kategorie und Kantennummer und rechts steht der Endknoten der inaktiven Kante.

3.4 Installation und Umgebung

Dieser Abschnitt beschreibt, wie der Parser geladen wird und welche Dateien und Ressourcen er benötigt. Ein Verzeichnis aller Dateien befindet sich im Anhang B. Wenn bei der Installation eigene Änderungen nötig sind, sollte man sich am Format der bisherigen Eintragungen orientieren.

Die Ladedatei `load-parser.lisp` des Parser enthält die Installation, d.h. alle nötigen Anpassungen an das Rechnersystem. Die Datei definiert das Package `chart-parser` mit Nickname `chp` und ruft die Funktion `load-module` auf. Der Aufruf enthält eine Liste aller Dateien, die geladen werden. Steht hinter der Datei `t`, dann verwendet ChaPLin kompilierten Code, bei `nil` lädt er den Quellcode. Hier kann man auch eigene Dateien hinzufügen.

`load-module` kompiliert die angegebenen Dateien bei Bedarf und legt die kompilierten Dateien (Binaries, Fasls, ...) im Unterverzeichnis `bin` ab. Die kompilierten Dateien sind für unterschiedliche Lispimplementierungen, Betriebssysteme oder Rechner (Prozessoren) inkompatibel. Möchte man im gleichen Dateisystem mit verschiedenen Plattformen arbeiten, dann definiert man jeweils einen Plattformnamen und eine Endung für die kompilierten Dateien in der Variable `*binary-extension*`. Der Plattformname ist das letzte Argument von `load-module`. Man definiert ihn am besten abhängig von der aktuellen Plattform mit der `#+`-Syntax von Common Lisp [Steele 90].

Die Dateien werden in der Reihenfolge geladen, in der sie in der Dateiliste stehen. Die Grammatikdateien und eigene Erweiterungen sollten erst nach dem Parserkern geladen werden. Zu Beginn einer Datei kann man mit `defmod`

Information über das Modul ablegen. `defmod` erhält als erstes Argument ein Schlüsselwort als Bezeichner für das Modul, anschließend zwei Strings, den Namen und das Datum des Moduls. Die Funktion `chp-version` druckt eine Übersicht über alle geladenen Module.

```
<USER.6> (chp-version)
ChaPLin Chart Parser
G. Burkert, M.Loethe
 Version 3.2  6-FEB-95

Parser Version 3.2, 6-FEB-95
Grammar types cf sf 3.2.2, 6-DEZ-95
Chart Parser Utilities, 2-FEB-95
Incremental Bottom Up Parsing 3.2, 6-DEZ-95
ATN Interpreter 1.2.1, AUG-94
ATN based line scanner 1.22, 24-FEB-93
Deutsche Beispielgrammatik :sf, 13-DEZ-95
Lexikonschnittstelle Beispiel, 13-DEZ-95
```

ChaPLin benötigt zur Arbeit nur den Parser, den Grammatiktyp der verwendeten Grammatik, die Grammatik selbst und ein passendes Lexikon. Die Utilities, den inkrementellen Parser und der Zeilenscanner muß man nur laden, wenn man sie benötigt. Der ATN-Interpreter wird vom Zeilenscanner benutzt.

Kapitel 4

Aufbau und Arbeitsweise von ChaPLin

Während man die in Kapitel 3 beschriebenen Funktionen ohne tiefere Kenntnis des Systems anwenden kann, muß man sich, um auch die fortgeschritteneren Möglichkeiten von ChaPLin zu nutzen, genauer mit dessen Arbeitsweise und damit auch mit Aspekten der Implementation vertraut machen. Daher beschreibt dieses Kapitel gleichzeitig die Arbeitsweise von ChaPLin und die Grundzüge der Implementation.

4.1 Der Einfluß des Grammatiktyps

Entwirft man eine rein kontextfreie Grammatik zur Verarbeitung natürlicher Sprache, so erhält man – wenn man die vielfältigen Besonderheiten und Varianten des Satzbaus nicht ausschließen möchte – eine Grammatik, die einen großen Teil der verfügbaren syntaktischen Information des Satzes nicht verwendet. Die verschiedenen aus der Literatur bekannten Grammatikformalismen haben die Aufgabe, diese Information in Form von sogenannten Features zur Steuerung des Parsers nutzbar zu machen. Die Repräsentation eines Grammatikformalismus in ChaPLin heißt *Grammatiktyp*. Ein wesentliches Element von ChaPLin ist die Trennung zwischen dem Kern des Parsers und den zum Grammatiktyp gehörenden Anteilen. Der Grammatiktyp bestimmt grundlegende Eigenschaften aller Analysephasen des Parsers.
Wie in Kapitel 2 beschrieben, erzeugt ChaPLin eine Chart, in der alle mit der gegebenen Grammatik möglichen Ableitungen als Kanten abgelegt werden. Die Kanten führen für den Bedarf des Parsers und für die Anwendung bei der Sprachverarbeitung Information folgender Art mit:

- Die zentrale Information in der Chart ist vom Grammatiktyp unabhängig und wird daher vom Kern des Parsers verwaltet. Dazu gehören z.B. Anfangs- und Endknoten von Kanten und deren Kategorien.

- Die o.g. Features werden mit grammatiktypabhängigen Regeln behandelt. Sie beeinflussen ebenfalls den Parsevorgang.

- Die Attribute nehmen dagegen am Parsevorgang nicht teil. Sie werden vom Grammatiktyp definiert. Die Attributinformation wird erst bei der Generierung der Ausgabestruktur berechnet. Die Attribute können zur weiteren Analyse verwendet, oder einfach ausgegeben werden.

Ein Grammatiksymbol im Sinne der Theorie der formalen Sprachen entspricht damit einer Kombination von Kategorie und Features. Die Attribute gehören nicht dazu, da sie den Parsevorgang nicht beeinflussen. Die Datenstrukturen für diese Information werden in Abschnitt 4.2 beschrieben.
Algorithmus 2 (S. 12) definiert die Analysephasen des Parsers. Da das Erstellen von Grammatiktypen Kenntnisse über die Arbeitsweise von ChaPLin erfordert, erläutern die folgenden Abschnitte die einzelnen Analysephasen und deren Grammatiktypoptionen gemeinsam. Folgende Bereiche des Parsers werden vom Grammatiktyp beeinflußt:

- Bei der syntaktischen Analyse definiert der Grammatiktyp die Featurebehandlung. Die Arbeitsweise der syntaktischen Analyse wird in Abschnitt 4.3 beschrieben.

- Einen besonders tiefgreifenden Einfluß hat der Grammatiktyp auf den Ausgabegenerator, der in Abschnitt 4.4 beschrieben wird. Der Grammatiktyp definiert die Attribute der Lesartkanten, ihre Berechnungsvorschriften und die Gestalt der semantischen Aktionen der Regeln.

- Die Schnittstelle zwischen Parser und Lexikon beschreibt Abschnitt 5.3. Auch für Feature- und Attributwerte benötigt der Parser (grammatiktypspezifische) Lexikoninformation.

- Da die Grammatikregeln Angaben zu Featurebedingungen und zur Attributierung enthalten, ist ihr Format vom Grammatiktyp abhängig. Der Ladevorgang für Grammatiken wird in Abschnitt 5.2 erläutert.

Die Arbeit der Parserfunktionen wird durch ihre Parameter gesteuert. Zu den Parametern in ChaPLin gehören einfache Optionen wie der Modus aber auch zusammengesetzte *Spezifikationen*. Spezifikationen in ChaPLin sind generell Listen aus Keywords und deren Werten. Die Menge der zulässigen Spezifikationsattribute ist grundsätzlich nicht beschränkt und ihre Reihenfolge ist nicht festgelegt. Spezifikationsattribute, die ChaPLin nicht kennt, werden einfach ignoriert. Spezifikationswerte können auch Funktionen sein, die der Parser dann an bestimmter Stelle aufruft.
Die Interaktion zwischen dem Grammatiktyp und dem Kern von ChaPLin ist mit folgenden Techniken realisiert:

- Der Grammatiktyp definiert Funktionen, die der Parserkern an bestimmten Stellen aufruft.

- Einige dieser Funktionen werden erzeugt, indem Codestücke des Grammatiktyps in eine Funktionsschablone des Parsers eingesetzt werden.

- Die Funktionen des Grammatiktyps verwenden Grundfunktionen des Parsers und steuern diese über Parameter (z.B. Spezifikationen).
- Manche Funktionen des Grammatiktyps erhalten vom Benutzer eingegebene Spezifikationen als Parameter. Da das Datenformat für Spezifikationen erweiterbar ist, kann der Grammatiktyp so eigene Optionen für die Benutzerfunktionen definieren.

4.2 Die Chart

Die Chart ist die grundlegende Datenstruktur des Parsers und somit für alle Analysephasen wichtig. Ihre Eigenschaften sind am Anfang von Kapitel 2 beschrieben. Sie ist als abstrakter Datentyp realisiert. Alle Zugriffe sollten über die festgelegten Zugriffsfunktionen erfolgen, da diese durch Fehlermeldungen abgesichert sind. Intern ist die Chart durch einen Vektor von Knoten (`*vertices*`) und einen Vektor von Kanten (`*edges*`) repräsentiert.

4.2.1 Knoten

Die Knoten der Chart stehen für die Zwischenräume im Satz. Die Numerierung beginnt mit `*left-vertex* = 1`. Mit `(get-vertex <nummer>)` greift man auf einen Knoten zu.

`vertex` Struktur

Ein Knoten ist eine Struktur `vertex` mit folgenden Komponenten:

`active-in` Liste der aktiven Kanten, die an diesem Knoten enden

`active-out` Liste der aktiven Kanten, die von diesem Knoten ausgehen

`number` Nummer des Knotens (entspricht dem Index in `*vertices*`)

`inactive-in` Liste der inaktiven Kanten, die an diesem Knoten enden

`inactive-out` Liste der inaktiven Kanten, die von diesem Knoten ausgehen

`tree-in` Liste der Lesartkanten, die an diesem Knoten enden

`tree-out` Liste der Lesartkanten, die von diesem Knoten ausgehen

Die Funktion `create-vertex` erzeugt einen neuen Knoten mit der nächsten freien Nummer. Die anderen Slots werden mit `nil` initialisiert. Die Funktion `get-vertex` mit Argument `n` liefert den Knoten mit der Nummer `n`.

4.2.2 Kanten

Es gibt drei Arten von Kanten: inaktive Kanten, aktive Kanten und Lesartkanten. Eine Kante wird durch einen Aufruf des Makros `insert-edge` erzeugt, wobei der Kantentyp angegeben wird. Auf Kanten kann mit `get-edge` über ihre Nummer zugegriffen werden.

4.2.2.1 gemeinsame Information

Alle Kantentypen haben die Graphinformation und einen Teil der syntaktischen Information gemeinsam.

`edge` Struktur

Die Basisstruktur `edge` hat folgende Slots:

`number` Nummer der Kante

`left` Nummer des Anfangsknotens.

`right` Nummer des Endknotens.

`cat` Kategorie der Kante

`contents` enthält je nach Kantentyp:

aktive Kante: Liste der bereits überspannten Kanten

syntaktische Terminalkante: Lexikoneintrag

syntaktische Nichtterminalkante: Bei einer mehrdeutigen Grammatik können verschiedene Ableitungswege auf das selbe Symbol führen. Durch Kantenverschmelzung hat eine Kante in diesem Fall mehrere verschiedene Inhalte. Damit man zur Berechnung der Lesartkanten die Ableitungswege wieder rekonstruieren kann, besitzt eine syntaktische Nichtterminalkante für jeden Ableitungsweg einen Wegeintrag bestehend aus der angewendeten Regel und den enthaltenen Kanten. Der Inhalt der Kante hat dann folgende Form:

`<contents> ::= ((<rule><edge>+) ...)`

terminale Lesartkante: Lexikoneintrag

nichtterminale Lesartkante: Liste der bereits überspannten Kanten.

`features` Features der Kante entsprechend der Definition im Grammatiktyp

4.2.2.2 Inaktive Kanten

`inactive-edge` Struktur

Inaktive Kanten sind Strukturen von Typ `inactive-edge`. Sie erben von `edge` und enthalten zusätzlich einen Slot für die zur syntaktischen Kante gehörenden Lesartkanten.

`trees` Eine Liste der Lesartkanten dieser Kante oder `:uncomputed`.

Zu Beginn enthält der Slot `trees` den Wert `:uncomputed`. Der Ausgabegenerator berechnet die Lesartkanten dann, wenn er sie das erste Mal benötigt und legt sie im Slot ab. Später verwendet er die im Slot abgelegten Lesartkanten.

Der Parser bietet die Möglichkeit an, Lesarten zu filtern, d.h Lesarten nach bestimmten Kriterien aus der Lesartenliste zu streichen. Verwendet man den Filter, so besitzt eine syntaktische Kante möglicherweise keine Lesarten und der Wert des Slots ist `nil`. Deswegen unterscheidet man die Fälle `nil` und `:uncomputed`.

4.2.2.3 aktive Kanten

Aktive Kanten sind Strukturen vom Typ `active-edge` und enthalten weitere Information über den augenblicklichen Zustand der Regelanwendung.

`active-edge` Struktur

Aktive Kanten fügen den von `edge` ererbten Slots daher noch folgende Slots hinzu:

`contents-name` Liste von Kategorien der Kanten, die die Kante überspannt

`needed` Liste von Kategorien, die eine aktive Kante noch benötigt.

`needed-name` Obige Liste, wobei gleichnamige Kategorien indiziert sind. Zum Beispiel lautet die Regel NP $\longrightarrow$ NP PP mit indizierten Kategorien NP.1 $\longrightarrow$ NP.2 PP. Featuremechanismen benötigen manchmal einen Index zur eindeutigen Identifizierung.

`needed-cond` Featurebedingungen für noch benötigte Kanten

`rule` Die gerade untersuchte Regel

4.2.2.4 Lesartkanten

Die Struktur für die Lesartkanten wird vom jeweiligen Grammatiktyp definiert. Dabei wird der Basistyp für Lesartkanten `tree-edge` um Slots für die vom Grammatiktyp verwendeten Attribute erweitert. Attribute werden erst in der Ausgabephase erzeugt und nehmen nicht an der syntaktischen Analyse teil (siehe Abschnitt 4.4)

`tree-edge` Struktur

Der Basistyp `tree-edge` erbt von `edge` und hat einen zusätzlichen Slot für die angewendete Regel.

`rule` Bei nichtterminalen Lesartkanten steht hier die angewendete Regel, sonst `nil`.

4.2.3 Agenda

Die Agenda ist zwar nicht direkt Bestandteil der Chart, gehört aber zu den grundlegenden Datenstrukturen von ChaPLin. Jedes Zusammentreffen einer aktiven und einer inaktiven Kante, d.h. jede Möglichkeit zur Anwendung

der Grundregel ist eine Konfiguration, die einmal betrachtet werden muß. Die beiden Kanten werden auf einem Stack, der Agenda, abgelegt. Die Agenda ist als Vektor mit einem `fill-pointer` implementiert.

4.3 Syntaktische Analyse

Dieser Abschnitt beschreibt, welche Arbeitsschritte es beim Chartaufbau und bei der syntaktischen Analyse gibt und wie dabei die Slots der Kanten belegt werden.

4.3.1 Kantenverschmelzung

ChaPLin erzeugt eine Chart, eine Datenstruktur, die alle mit der gegebenen Grammatik möglichen Ableitungsbäume enthält. Grundprinzip jedes Verfahrens zur Berechnung des Syntaxbaums ist, festzustellen, ob man durch Anwendung einer Grammatikregel eine bestimmte Folge von Symbolen durch ein Nichtterminalsymbol ersetzen kann. Grammatiksymbole entsprechen in einem Chartparser inaktiven Kanten. Für eine mögliche Regelanwendung werden aktive Kanten angelegt. Solche aktiven Kanten können nach der Grundregel inaktive Kanten (Symbole) lesen, und es wird dann eine neue, verlängerte Kante angelegt.

Bei der syntaktischen Analyse unterscheidet der Parser Kanten nur nach Kategorie und Features. Die Attribute sind für die syntaktische Analyse unwesentlich. Versucht der Parser, eine inaktive Kante einzufügen, die nach Kategorie und Features äquivalent zu einer existierenden inaktiven Kante ist, so *verschmilzt* er diese Kanten. Die unterschiedlichen Ableitungswege werden aber protokolliert, denn es kann später – bei der Ausgabegenerierung – für unterschiedliche Ableitungswege verschiedene Attributwerte geben. Die erfolgreichen Ableitungswege expandiert der Parser bei der Ausgabegenerierung wieder.

Der Grund für dieses komplizierte Vorgehen liegt in der Effizienz. Sei n die Anzahl der Eingabeelemente und g die Anzahl der Grammatiksymbole. In den folgenden Abschätzungen modelliert g den schlimmsten Fall der Mehrdeutigkeit. Die Anzahl der Grammatiksymbole g ist bei Grammatiken für natürliche Sprache verhältnismäßig groß, da man die verschiedenen zulässigen Featurewerte jeweils als verschiedene Grammatiksymbole berücksichtigen muß. Bei realen Sätzen und Grammatiken ist jedoch die Mehrdeutigkeit der Ableitung – nämlich die Anzahl paralleler Kanten mit verschiedener Kategorie und Featurewerten für einen Teil der Chart – ziemlich klein. Daher kann man g als kleinen konstanten Faktor betrachten.

- Bei einem Verfahren mit Kantenverschmelzung ist die Anzahl der möglichen Kanten durch $\frac{1}{2}gn^2$ beschränkt, da jeder Knoten nur mit Knoten größerer Nummer verbunden ist und alle parallelen Kanten verschieden sind.

- Verzichtet man auf Kantenverschmelzung, so kann die Kantenzahl exponentiell wachsen. Dieser Fall tritt in der Praxis auf, wenn die Grammatik einen stark mehrdeutigen Anteil enthält, z.B. für Aufzählungen.

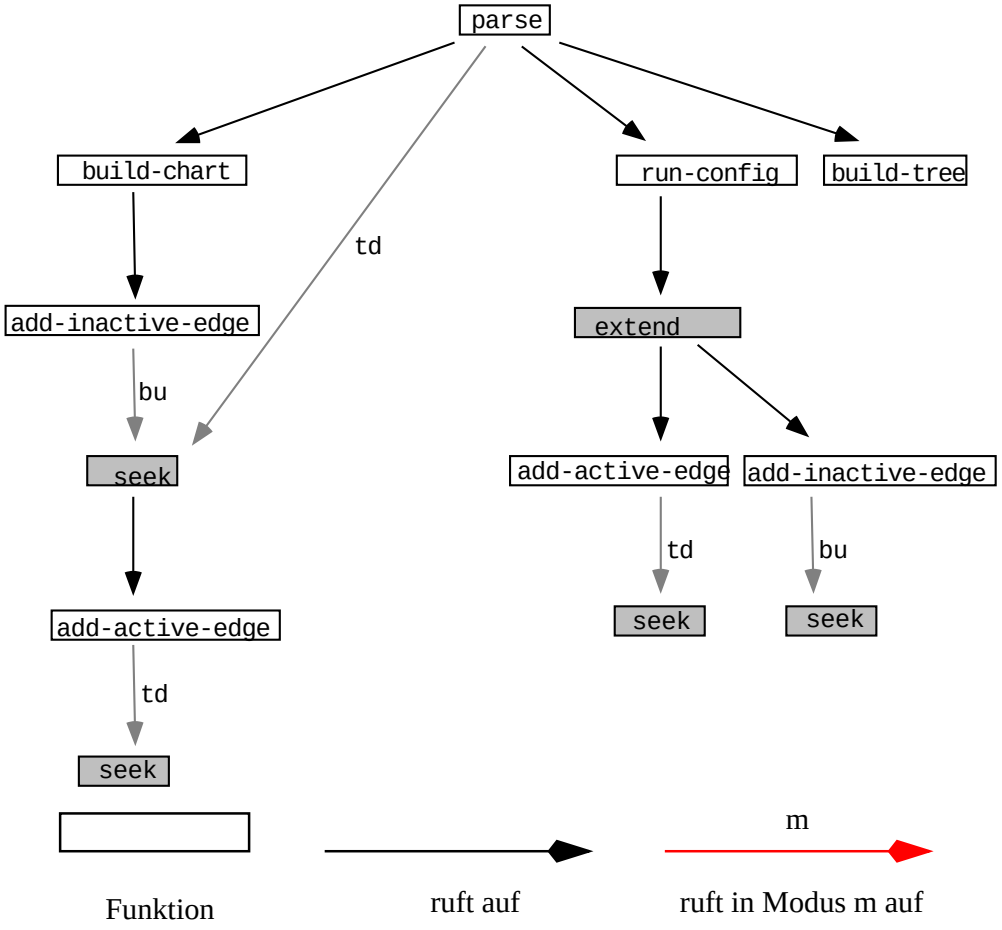

Abbildung 4.1: Aufrufstruktur von ChaPLin

Daher arbeitet ChaPLin mit Kantenverschmelzung. Die Kantenverschmelzung reduziert nicht die Zahl der Lesarten, sondern ist nur eine kompakte Darstellung des Parsewalds. Das exponentielle Wachstum der Chart tritt dann möglicherweise auf, wenn man die Lesarten in der Chart expandiert. Man kann aber die Lesarten mit `:output :count` vor der Expansion zählen und bei problematischen Sätzen auf die Expansion verzichten. Die Kantenverschmelzung ermöglicht es so, Problemfälle abzufangen.

4.3.2 Die Arbeitsschritte der syntaktischen Analyse

Dieser Abschnitt beschreibt die einzelnen Arbeitsschritte aus Algorithmus 1 (S. 7) genauer und nennt die zugehörigen Funktionen von ChaPLin. Abbildung 4.1 zeigt eine vereinfachte Fassung des Aufrufbaums von ChaPLin. Bei der Definition eines Grammatiktyps gibt man 4 Behandlungsregeln für Features an. Sie definieren das Verhalten beim Lesen eines Symbols, beim Anwenden einer Grammatikregel, einen Äquivalenztest und einen Anwendbarkeitstest und werden bei der Beschreibung der Grammatiktypoptionen genauer erläutert. Die

Defaultwerte entsprechen der Verwendung einer kontextfreien Grammatik ohne Berücksichtigung von Features.
Die Zuweisungen und Bedingungen sind in Pseudocode notiert. Bei den Kantenzugriffen wird `edge-` weggelassen. Die grammatiktypabhängigen Featurebehandlungsregeln werden mit einem * gekennzeichnet.

4.3.2.1 Terminalsymbol eintragen

Zu Beginn des Parsens werden die Knoten angelegt und die Lexeme als inaktive Terminalkanten in die Chart eingetragen. Für jeden Lexikoneintrag L erzeugt ChaPLin also eine inaktive Kante I.

Funktion `add-input-item`

Bedingung keine

Zuweisungen

cat	:=	(cat L)
features	:=	(features L)
contents	:=	L

4.3.2.2 Aktivieren

Beim Aktivieren wird für jede anwendbare Regel R eine aktive Kante A1 in die Chart eingetragen. A1 ist eine Schlinge (endet an ihrem Anfangsknoten), da sie noch kein Symbol (inaktive Kante) gelesen hat.

Funktion `seek`

Bedingung Je nachdem ob bottom-up oder top-down geparst wird, aktiviert der Parser die Chart zu unterschiedlichen Zeitpunkten (vgl. Kapitel 2.2). Daher auch folgende unterschiedliche Bedingungen:

top-down Es muß eine aktive Kante A2 existieren, mit

(right A2)	=	(left A1)
(rule-lhs R)	=	(first (needed A2))

bottom-up Es muß inaktive Kante I existieren, mit

(left I)	=	(left A1)
(first (rule-rhs R))	=	(cat I)

Diese Bedingung stellt die Anwendbarkeit der Regel fest. Die Features werden dabei nicht berücksichtigt. Der Parser trägt Schlingen, deren Features nicht zur Kante passen, erst einmal in die Chart ein. Er kann sie aber später nicht mit der Grundregel verlängern.

Wenn die Chart an diesem Knoten schon einmal für die Regelkategorie aktiviert worden ist, dann wird die Aktivierung nicht noch einmal ausgeführt. Dadurch werden Probleme mit linksrekursiven Regeln vermieden.

Die Chart ist bereits aktiviert, wenn an dem Knoten eine aktive Schlinge A3 folgender Form existiert:

top-down	(rule-lhs R)	=	(cat A3)
bottom-up	(first (rule-rhs R))	=	(car (needed A3))

Zuweisungen

(cat A1)	:=	(lhs R)
(features A1)	:=	derzeit: ()
(contents A1)	:=	()
(needed A1)	:=	(rhs R)
(needed-cond A1)	:=	(rhs-features R)
(rule A1)	:=	R

Damit unterscheiden sich die beiden Modi top-down und bottom-up nur in den Aktivierungsbedingungen und im Zeitpunkt, an dem die Funktion `seek` aufgerufen wird.

4.3.2.3 Symbol lesen ohne Regel zu vervollständigen

Bei der Anwendung der Grundregel wird die aktive Kante A1 durch die inaktive Kante I verlängert, wenn diese beiden Kanten passen. Falls der Rumpf der Regel von A1 danach noch weitere Symbole benötigt, wird eine aktive Kante A2 erzeugt.

Funktion `extend`

Bedingung

(cat I)	=	(first (needed A1)
(match-features* A1 I)	≠	NIL

Zuweisungen

(cat A2)	:=	(cat A1)
(features A2)	:=	(act-features* A1 I)
(needed A2)	:=	(rest (needed A1))
(needed-cond A2)	:=	(rest (needed-cond A1)
(contents A2)	:=	(cons (cat I) (contents A1))
(rule A2)	:=	(rule A1)

4.3.2.4 Symbol lesen, Ableitung hinzufügen

Wie vorhin wird die aktive Kante A mit einer inaktiven Kante I1 verlängert. Wenn dabei für jedes Symbol im Rumpf der Regel von A eine inaktive Kante gelesen wurde, dann wendet der Parser die Regel an und erzeugt die inaktive Kante I2.

Funktion `extend`

Bedingung

(cat I1)	=	(first (needed A1)
(match-features* A1 I)	≠	NIL

Zuweisungen

```
(cat I2)        :=  (cat A)
(features I2)   :=  (inact-features* A I1)
(contents I2)   :=  (list (cons  (rule A)
                                 (reverse (cons  (contents I1)
                                                 (contents A)))))
```

4.3.2.5 Symbol lesen, Ableitungsweg hinzufügen

Ebenso wie im vorigen Fall wird nach der Grundregel die aktive Kante A mit einer inaktiven Kante I1 verlängert und die Grammatikregel ist anwendbar. Eine Kante I2 mit der gleichen Kategorie wie die entstehende Kante und äquivalenten Featurewerten existiert aber schon. In diesem Fall führt ChaPLin eine Kantenverschmelzung durch und trägt nur eine weitere Lesart für die Kante I2 ein.

Funktion `extend`

Bedingung

```
(cat I)                      =   (first (needed A1)
(match-features* A1 I1)      ≠   NIL
(cat A)                      =   (cat I2)
(fea-equalp* (inact-features* A I1)(features I2))
```

Zuweisungen

```
contents(I2)  :=  (cons (cons (rule A)
                              (reverse (cons (contents I1)
                                             (contents A))))
                        (contents I2))
```

4.3.3 Grammatiktypoptionen für die syntaktische Analyse

Bei der Erstellung eines Grammatiktyps definiert man die oben mit einem * markierten 4 Zuweisungsregeln und Bedingungen für Features. Die Defaultwerte führen zu einer rein kontextfreien Grammatik ohne Verwendung von Featurewerten.

Die Funktion `extend-config` baut bei der Definition des Grammatiktyps die Grundregelfunktion `extend` aus der Funktionsschablone und 3 Codestücken zusammen. Weil die Grundregel für jede Konfiguration aufgerufen wird, muß man bei ihr auf Effizienz achten.

Die Grundregelfunktion erhält als Parameter die beiden Kanten der Konfiguration `act` und `inact`, den Modus `mode` und die indizierte Regelmenge `rules` . Diese Variablen kann man in den Codestücken verwenden. Sinnvolle Parameter für die Codestücke des Featuremechanismus sind normalerweise nur die in der aktiven Kante abgelegte aktuelle Regel, die Features `(edge-features act` bzw. `inact)` und eventuell noch die Kategorien `(edge-cat act` bzw.`inact)` der beiden Kanten `act` und `inact`.

act-features Lesevorgang ohne Regelanwendung

Dieses Codestück zum Einbau in die Grundregelfunktion berechnet bei der Erzeugung einer aktiven Kante deren Features, wobei die in in der aktiven Kante abgelegte aktuelle Regel, die Features der beiden Kanten `act` und `inact` und eventuell noch deren Kategorien sinnvolle Parameter sind.

Default: `()` – bei kontextfreien Grammatiken sind alle Features `nil`.

inact-features Lesevorgang mit Regelanwendung

Dieses Codestück zum Einbau in die Grundregelfunktion berechnet die Features bei der Erzeugung einer inaktiven Kante und hat ebenfalls Zugriff auf die Parameter der Grundregelfunktion. Jedoch sind auch hier üblicherweise nur die aktuelle Regel, die Features und eventuell noch die Kategorien der beiden Kanten sinnvoll.

Default: `()` – bei kontextfreien Grammatiken sind alle Features `nil`.

fea-equalp Äquivalenztest für Kantenverschmelzung

Diese Vergleichs*funktion* erhält die Features zweier inaktiver Kanten mit gleicher Kategorie als Argumente. Wenn die Funktion `nil` zurückgibt, legt ChaPLin eine eigene Kante an, sonst werden die Kanten verschmolzen.

Default: `#'eq` – erlaubt die Verschmelzung immer, da im kontextfreien Fall alle Features `nil` sind.

match-features Anwendbarkeitstest

Codestück zum Einbau in die Grundregelfunktion. Es prüft, ob die Features der inaktiven Kante zu denen der aktiven Kante passen, d.h. zulassen, daß die aktive Kante um die inaktive Kante verlängert wird. Ist dies nicht der Fall, muß das Codestück `nil` ergeben, sonst einen anderen Wert.

Default: `t` – bei kontextfreien Grammatiken gibt es keine zusätzliche Restriktion durch Features.

4.4 Ausgabegenerierung

Die letzte Analysephase aus Algorithmus 2 (S. 12) ist die Ausgabegenerierung. Der folgende Algorithmus teilt diese Phase noch weiter auf.

Algorithmus 3

1. Expandiere die Ausgabespezifikation mit der Grammatiktypfunktion `build-fn`. Algorithmus 7 (S. 60) beschreibt das Vorgehen für den Grammatiktyp `:sf`.

2. Enthält die Spezifikation das Attribut `:result`, dann ist der Rückgabewert bereits bestimmt. Gib ihn zurück.

3. Sonst bestimme die erfolgreichen Kanten.
   (a) Wenn vollständige Parses vorhanden sind, verwende diese.
   (b) Sonst, wenn gewünscht, suche Kantenfolgen in der Chart als Teilergebnis (partial parsing).
4. Wenn eine Ausgabeform ohne Parsebäume gewünscht wird, erzeuge sie und gib sie zurück.
5. Erzeuge die Lesartkanten zu jeder inaktiven Kante und filtere sie, d.h. streiche anhand einer Bedingung Lesarten.
6. Erzeuge aus den Lesartkanten die Knoten und Blätter des Parsebaums.
7. Führe eine Nachbearbeitung auf oberster Ebene der Liste der Parsebäume durch.

Die Arbeitsweise dieser Unterphasen wird jetzt im einzelnen vorgestellt.

4.4.1 Die Ausgabespezifikation

Die Ausgabespezifikation ist eine Liste von Keywords und deren Werten. Die Funktionen des Ausgabegenerators greifen auf die Schlüsselwörter mit `getf` zu. Es ist daher jederzeit möglich, weitere Keywords für grammatiktypspezifische Erweiterungen einzuführen, da Keywords, die der Ausgabegenerator eines anderen Grammatiktyps nicht kennt, einfach ignoriert werden.
Für einfache Fälle sind im Grammatiktyp Keywords als Abkürzungen für Spezifikationen definiert, die vom Spezifikationsübersetzer `build-fn` des Grammatiktyps in die detaillierte Spezifikation übersetzt werden. Die Abkürzung `:succ` wird z.B. in `'(:succ t)` expandiert. Wenn der Übersetzer den Rückgabewert ohne Zugriff auf Funktionen des Parserkerns berechnet, gibt er `'(:result <Wert>)` zurück. Der Ausgabegenerator gibt dann diesen Wert aus.

4.4.2 Suche nach erfolgreichen Kanten

Nachdem die Spezifikation expandiert ist, beginnt die eigentliche Ausgabegenerierung. Im ersten Schritt untersucht der Ausgabegenerator, ob der Parselauf erfolgreich war und wenn ja, welche Kanten erfolgreich sind. Normalerweise sind die Kanten erfolgreich, die die ganze Chart überspannen. Ist der Wert des Attributs `:last t`, dann sind es alle Kanten, die den letzten Knoten als Endknoten haben.
Wenn das Parsing nicht erfolgreich ist, dann wird nach Fragmentfolgen durch die Chart gesucht.
Die Suche nach den erfolgreichen Kanten übernimmt die Funktion `success-edges` nach folgendem Verfahren:

Algorithmus 4

1. Zuerst bestimmt der Parser die Menge der Kanten mit den richtigen Anfangs- und Endknoten. Normalerweise sind das der erste und der letzte Knoten der Chart. Bei der Option `:last` ist jedoch der Anfangsknoten beliebig und bei der Erzeugung partieller Bäume werden die entsprechenden Grenzen des Chartfragments verwendet.

2. Aus dieser Kantenmenge werden die ignore-Kanten gestrichen, damit das Ergebnis so aussieht, als gäbe es diese Kategorien nicht.

3. Kanten, die nicht einer Startkategorie angehören, werden gestrichen.

4. Wenn eine Kante in dieser Menge in einer anderen Kante der Menge enthalten ist, wird sie gestrichen.

Damit sind die erfolgreichen Kanten bestimmt.

4.4.3 Einfache Ausgabeformen

Nachdem die Menge der erfolgreichen Kanten bekannt ist, wird der Rückgabewert für die einfachen Ausgabeformen, die ohne Erzeugung eines Parsebaums auskommen, bestimmt. Bei der Spezifikation `:succ t` wird `nil` zurückgegeben, wenn die Kantenmenge leer ist, sonst `t`. `:cat t` erzeugt eine Liste der abgeleiteten Kategorien. Enthält die Spezifikation `'(:count t)`, gibt ChaPLin die Anzahl der Lesarten zurück.

4.4.4 Erzeugung der Lesarten

Die Chart ist eine kompakte Darstellung des Parsewalds, denn durch die Kantenverschmelzung können mehrere Ableitungswege auf dieselbe inaktive Kante führen. Nach den Bedingungen für die Kantenverschmelzung sind für diese Wege die Features äquivalent und die Kategorie gleich. Nun möchte man aber für die Ausgabe weitere Werte, die Attribute, berechnen. Auf den unterschiedlichen Ableitungswegen zu einer Kante können sich jedoch unterschiedliche Attributwerte ergeben. Die Attribute werden vor der Baumerzeugung berechnet und in Lesartkanten abgelegt und damit bei den Lesarten die Kantenverschmelzung wieder expandiert.
Die Menge aller Lesarten für eine syntaktische Kanten wird im Slot `trees` abgelegt, so daß der Parser die Lesarten nur einmal berechnen muß, wenn eine syntaktische Kante mehrmals in einem Parsebaum verwendet wird. Das Verfahren verwendet Funktionen des Grammatiktyps, die in Abschnitt 4.4.7 beschrieben werden. Die Lesarterzeugung für eine syntaktische Kante läuft folgendermaßen ab:

Algorithmus 5
wenn die Lesarten der syntaktischen Kante bereits berechnet sind

dann gib sie zurück

sonst
1. Für jeden Inhalt der syntaktischen Kante
   - Bei einem terminalen Inhalt rufe die Funktion `term` des Grammatiktyps.
   - Bei einem nichtterminalen Inhalt
     (a) Berechne die Lesarten der Kanten des Inhalts (rekursiv).
     (b) Bilde das kartesische Produkt der Lesartmengen der enthaltenen Kanten. Damit werden die Mehrdeutigkeiten der Unterbäume weiterpropagiert.
     (c) Konstruiere für jedes Tupel aus dem kartesischen Produkt die Lesart mit der Funktion `nonterm` des Grammatiktyps.
2. Vereinige die Lesartmengen der Inhalte. Dieser Schritt expandiert die Kantenverschmelzung.
3. Filtere diese Menge mit der Filterfunktion `filter` des Grammatiktyps.
4. Trage das Ergebnis in den Slot `trees` der syntaktischen Kante ein und gib es zurück.

Die Terminalfunktion `term` entnimmt die Attributwerte für die Lesartkante aus dem Lexikoneintrag. Der Nichtterminalfunktion `nonterm` stehen die enthaltenen Lesartkanten und die Grammatikregel zur Verfügung. Es handelt sich damit um synthetisierte Attribute (zusammengesetzte Attribute). Die Filterfunktion `filter` kann Lesarten anhand von Attributwerten entfernen und die Lesartenliste umsortieren.

4.4.5 Baumerzeugung

In der Syntaxbaumerzeugungsphase erzeugt der Parser zu jeder erfolgreichen Lesartkante den zugehörigen Syntaxbaum, indem er die Inhaltshierarchie der Lesartkante durchwandert und dabei eine Präfixnotation des Parsebaums aufbaut. Die Erzeugung der einzelnen Knoten wird von der Baumspezifikation gesteuert, die unter `:tree` in der Ausgabespezifikation abgelegt ist. Die Baumspezifikation enthält Werte zu den drei Schlüsselwörtern `:lex`,`:struct` und `:node`. Diese Werte sind Muster für die Knoten des jeweiligen Teils des Parsebaums und beschreiben, welche Information aus der Lesartkante in den Parsebaum übernommen werden soll.

- Terminalkanten können sowohl Lexikoneinträge als auch Struktureinträge enthalten. Die `:lex`-Spezifikation steuert die Erzeugung von Parsebaumblättern aus Terminalkanten, die einen *Lexikon*eintrag enthalten.

- Die `:struct`-Spezifikation behandelt den Einbau eines Unterparsebaums für Terminalkanten, die einen *Struktur*eintrag enthalten. Mit diesem Verfahren kann ChaPLin Folgen von bereits analysierten Nichtterminalsymbolen parsen und einen Gesamtsyntaxbaum aufbauen.

- Die `:node`-Spezifikation steuert die Erzeugung eines Knotens für eine Nichtterminalkante. So entstehen die inneren Knoten und die Wurzel des Syntaxbaums.

Der Baumerzeugungsalgorithmus arbeitet die Knotenspezifikationen folgendermaßen ab:

Algorithmus 6

1. Ist die Spezifikation `NIL`, dann gib `NIL` zurück.

2. **Nur bei Terminalkanten:**
   Ist die Spezifikation = `:all`, so gib den ganzen Eintrag (Lexikon- oder Struktur-) zurück.

3. **Nur bei Terminalkanten:**
   Ist die Spezifikation ein Symbol und dieses Symbol ist als Attribut im Eintrag enthalten, so gib dessen Wert (das dem Symbol folgende Listenelement) zurück.

4. Ist die Spezifikation eine Liste, dann rufe diesen Algorithmus für jedes Element und gib die Liste der Ergebnisse zurück.

5. Ist die Spezifikation eine Funktion, so rufe sie mit der Lesartkante als Argument auf, und gib ihr Ergebnis zurück.

6. Ist die Spezifikation ein Symbol und eine Funktion mit diesem Namen existiert, dann rufe diese Funktion mit der Lesartkante als Argument auf.

7. Gib alle anderen Symbole unverändert zurück.

8. Jeder andere Wert führt zu einem Fehler.

4.4.6 Nachbearbeitung

Das Verfahren zur Baumerzeugung bearbeitet den Parsebaum rekursiv und unterscheidet nicht zwischen Wurzelknoten und inneren Knoten. Erst bei der Nachbearbeitung der Parsebäume kann man den Wurzelknoten eine andere Gestalt als den inneren Knoten geben. Die Funktion `tree-postproc` des Grammatiktyps erhält den erzeugten Baum, die Lesartkante und die Ausgabespezifikation als Argument. Diese Funktion kann z.B. dem Wurzelknoten eines Parsebaums weitere Attributwerte aus der Lesartkante hinzufügen.

Die Nachbearbeitung ermöglicht auch die Erzeugung von Struktureinträgen, das sind Parsebäume in einer Form, die der Parser in einem späteren Lauf als Terminalsymbole verwenden kann.

4.4.7 Grammatiktypoptionen der Ausgabegenerierung

Für den Ausgabegenerator gibt es ebenso wie für die syntaktische Analyse Einflußmöglichkeiten durch Grammatiktypoptionen. Der Grammatiktyp definiert bei Bedarf einen eigenen Lesartkantentyp als Spezialisierung von `tree-edge`. Dieser Typ erhält für jedes Attribut, das der Grammatiktyp berechnet, einen Slot.

In der Definition des Grammatiktyps gibt es folgende Optionen für den Ausgabegenerator:

`build-fn` Der Spezifikationsübersetzer ist eine Funktion, die beim Aufruf folgende Argumente erhält:

`grammar` Grammatik

`mode` Modus (`:bu` oder `:td`)

`find` Startkategorie

`type` Ausgabespezifikation

Die Funktion expandiert die Ausgabespezifikation für den Gebrauch des Ausgabegenerators (vgl. Abschnitt 4.4.1).

Default: Eine Funktion, die die Ausgabespezifikation `type` unverändert zuückgibt.

`edge-constr` Hier wird der Name der Konstruktorfunktion (Symbol) für die Lesartkanten des Grammatiktyps angegeben.

Default: `make-tree-edge`

`term-actions` Die Attributierungsregeln für Terminalsymbole sind eine Liste von Attributnamen als Keywords und Codestücken zur Berechnung der Attributwerte. Die Codestücke werden in einen Aufruf des Lesartkantenkonstruktors eingebaut, der immer dann gerufen wird, wenn die Lesartkante eines Terminalsymbols erzeugt wird. Die Slots der syntaktischen Kante übernimmt der Ausgabegenerator automatisch und für die Attribute wird der Wert des Codestücks benutzt.

Den Codestücken für die Terminalsymbole stehen zwei Variablen zur Verfügung:

`entry` Der Lexikoneintrag der syntaktischen Terminalkante.

`syntedge` Die syntaktische Terminalkante selber.

Default: `()`, d.h. es werden keine Attribute berechnet.

`nonterm-actions` Ebenso wie bei den `term-actions` gibt man die Attributierungsregeln für die Nichtterminalsymbole als Liste von Attributnamen und Codestücken zur Berechnung der Attributwerte an. Die syntaktischen Slots belegt der Ausgabegenerator automatisch.

Die `nonterm-actions` beschreiben, wie der Ausgabegenerator bei der Erzeugung der nichtterminalen Lesartkanten die Attributwerte berechnet. Eine Kante wird bei Kantenverschmelzung auf mehreren Wegen erreicht, wobei für jeden dieser Wege ist die Menge der enthaltenen Lesartkanten und die angewendete Regel charakteristisch sind. Bei zusammengesetzten Attributen hängen die Attributwerte nur vom Ableitungsweg ab, so daß der Ausgabegenerator den Codestücken folgende drei Variablen zur Verfügung stellt:

`tree-cont` Liste der bei der Ableitung verwendeten Lesartkanten. Da zusammengesetzte Attribute von den Attributwerten der Konstituenten abhängen, greift man auf die enthaltenen Lesartkanten zu und nicht auf die syntaktischen Kanten.

`rule` Die angewendete Regel wird übergeben, damit man regelspezifische Werte zur Attributberechnung mit heranziehen kann. Damit kann der Grammatiktyp die Möglichkeit anbieten, für ein Attribut regelspezifische Berechnungsvorschriften – sogenannte semantische Aktionen [Aho et al. 86] – zur Attributberechnung zu definieren.

`syntedge` Die syntaktische Terminalkante enthält Angaben wie Kategorie, Features, ... Diese Werte stehen der Attributberechnung ebenfalls zur Verfügung.

Default: `()`, d.h. es werden keine Attribute berechnet.

`filter` Der Lesartenfilter ist eine Funktion, die folgende Argumente erhält:

`edges` Die für eine syntaktische Kante erzeugten Lesartkanten.

`spec` Die Ausgabespezifikation.

Die Filterfunktion gibt eine Liste von Lesartkanten zurück, die dann als Lesarten für die syntaktische Kante eingetragen werden. Die Filterfunktion kann also Lesarten streichen und die Liste umordnen. Damit kann die Filterfunktion bei der Lesartgenerierung für jeden Ableitungsschritt unwahrscheinliche Lesarten ausschließen. Für weitere Ableitungsschritte berücksichtigt der Ausgabegenerator nur die zugelassenen Lesarten der Konstituenten.

Die Filterfunktion erhält die Ausgabespezifikation als Parameter, so daß man verschiedene Filteroptionen definieren kann, die man beim Aufruf des Parsers durch Angaben in der Ausgabespezifikation auswählt.

Default: Funktion, die die Kantenliste `edges` unverändert zurückgibt.

`tree-postproc` Die Funktion zur Parsebaumnachbearbeitung erhält den Parsebaum, die zugehörige Lesartkante und die Ausgabespezifikation als Argumente und gibt die ggf. geänderte Liste der Parsebäume zurück.

Default: Funktion, die je nach Ausgabespezifikation die Parsebäume unverändert zurückgibt oder daraus Struktureinträge erzeugt.

Kapitel 5

Sprachwissen

Dieses Kapitel beschreibt, wie Wissen über natürliche Sprache in ChaPLin eingebracht wird. Möchte man Grammatiken eines bestimmten Grammatikformalismus mit ChaPLin verwenden, definiert man einen Grammatiktyp (vgl. Abschnitt 5.1). Bisher sind für ChaPLin drei Grammatiktypen erstellt worden, der kontextfreie Grammatiktyp `:cf`, der Grammatiktyp `:sf` mit flachen Featurelisten und ein PATR-II-Unifikationsgrammatiktyp `:fu` [Schmidt 92]. Die Grammatiktypen `:cf` und `:sf` sind mit der Version 3.2 des Parsers lauffähig und werden in den Abschnitten 5.4 und 5.5 erläutert. Abschnitt 5.3 beschreibt die Schnittstelle zwischen dem Parser und einem Lexikon und Abschnitt 5.2 Definition und Zugriff auf die Grammatik.

5.1 Grammatiktyp

Der Grammatiktyp implementiert das Verhalten des Grammatikformalismus, indem er Funktionen für die syntaktische Phase des Parses, den Ausgabegenerator und den Grammatiklader zur Verfügung stellt. Die Optionen bei der Definition eines Grammatiktyps werden mit den zugehörigen Teilen des Parsers, die sie steuern, dokumentiert: die Optionen zur syntaktischen Phase in Abschnitt 4.3.3, die Optionen zum Ausgabegenerator in Abschnitt 4.4.7 und die Optionen des Grammatikladers am Ende des Abschnitts 5.2.

`*grammar-types*` Variable

Die globale Variable `*grammar-types*` enthält eine Liste der geladenen Grammatiktypen.

`grammar-type` Struktur

Jeder Grammatiktyp wird intern als eine Instanz der Struktur `grammar-type` abgelegt, die alle grammatiktypspezifischen Informationen enthält. Sie besteht aus folgenden Slots:

`key` Keyword, Name des Grammatiktyps. Der Grammatiktyp für kontextfreie Grammatiken heißt `:cf`, der für Grammatiken mit flachen Featurelisten `:sf`.

`rule-test` Prädikat, das testet, ob die Syntax einer Regel in diesem Grammatiktyp zulässig ist.

`create-rule` Diese Funktion erzeugt aus der Listennotation einer Grammatikregel ein Regelobjekt.

`extend` Funktion, die die Grundregel der aktiven Chartanalyse implementiert. Sie testet für eine Konfiguration, ob eine Erweiterung möglich ist und führt sie ggf. durch. Sie hat die Argumente:

`act` Die aktive Kante der augenblicklichen Konfiguration.

`inact` Die inaktive Kante der Konfiguration.

`mode` Modus: `:bu` für bottom-up oder `:td` für top-down.

`rules` Zugriffsfunktion auf die Grammatikregeln.

`build-fn` Diese Funktion expandiert die Ausgabespezifikation.

`term` Die Funktion erzeugt eine terminale Lesartkante.

`nonterm` Die Funktion erzeugt eine nichtterminale Lesartkante.

`filter` Die Funktion streicht unerwünschte Lesarten.

`tree-postproc` Die Funktion dient zur Nachbearbeitung der Parsebäume.

`def-grammar-type` Makro

`(def-grammar-type name &key [option]*)`

Dieses Makro definiert einen Grammatiktyp mit Namen `name` und legt ihn in der Variable `*grammar-types*` ab. Wenn ein Grammatiktyp mit diesem Namen schon existiert, wird dieser durch den gerade definierten Grammatiktyp ersetzt. Die Optionen werden bei den zugehörigen Teilen des Parsers erläutert.

Einen Grammatiktyp implementiert man durch eine Lispdatei, die folgende Elemente enthält:

- Die Definition des Lesartkantentyps als Spezialisierung von `tree-edge` , sofern man Attribute benutzt.
- Die Definition des Regeltyps als Spezialisierung von `rule` , falls der Ausgabegenerator Regelinformation benötigt.
- Die Definitionen aller für die Grammatiktypoptionen benötigten Funktionen und globalen Parameter.
- Einen Aufruf des Definitionsmakros `def-grammar-type`.

Diese Datei kann erst *nach* dem Parserkern geladen werden.

5.2 Grammatik

Die Defaultgrammatik befindet sich in der Variablen `*grammar*`.

`grammar` Struktur

Eine Grammatik wird als Element der Struktur `grammar` mit folgenden Slots abgelegt.

`rule-list` Regelmenge in Listennotation.

`access-td` Die Zugriffsfunktion für die Top-down-Aktivierung erhält die Zielkategorie als Argument. Bei der Top-down-Aktivierung benötigt man alle Regeln, die auf ein bestimmtes Zwischenziel hinführen (vgl. Abschnitt 2.2). Die Closure `access-td` gibt deswegen die Regelobjekte zurück, die die angegebene Kategorie als Regelkopf haben.

`access-bu` Die Zugriffsfunktion für die Bottom-up-Aktivierung erhält die Kategorie der gerade eingetragenen Kante als Argument. Man benötigt hier Regeln, deren Anwendung mit dem neuen Symbol beginnt. Daher gibt die Closure `access-bu` die Regelobjekte zurück, die die angegebene Kategorie als erstes Symbol im Regelrumpf haben.

`key` Der Name des Grammatiktyps der Grammatik.

`ignore` Liste von Ignore-Kategorien. Sie sind nur Hilfsschritte bei der Definition der Grammatik. Im Parsebaum werden diese Kategorien nicht als Unterbaum eingebaut, sondern durch ihren Inhalt ersetzt.

`start` Das Startsymbol der Grammatik. Beim Aufruf des Parsers kann durch den Parameter `:find` ein anderes Startsymbol angegeben werden.

`string-cats` String-Kategorien der Grammatik
Die Terminalsymbole der Grammatik sind im Normalfall nicht die Eingabeelemente (wie z.B. „Berg“) selbst, sondern Kategorien (z.B. N). In einigen Fällen hängen aber grammatikalische Konstruktionen von einem bestimmten Wort ab, wie z.B. die Satzreihung mit “und”. Dafür können Regeln das Wort direkt – als Stringkategorie – enthalten.

Eine Grammatik enthält Regeln. Extern (z.B. in Dateien) werden Regeln in einer grammatiktypabhängigen Listennotation angegeben. Intern speichert der Parser Regelobjekte als Instanzen der folgenden Struktur:

`rule` Struktur

Die Basisstruktur für Regeln enthält die Slots für die syntaktische Analysephase:

`lhs` Der Regelkopf (linke Seite) ist ein Symbol.

`rhs` Der Regelrumpf (rechte Seite) ist eine Liste von Symbolen. Hier kann auch die wild card `?` benutzt werden, die für eine beliebige Konstituente steht.

`rhs-name` Regelrumpf mit indizierten Kategorien.

`rhs-cond` Featurebedingungen. Die Regel wird nur angewendet, wenn diese Bedingungen erfüllt sind.

`result` Wenn der Parser eine Regel anwendet, erzeugt er eine inaktive Kante. Die Features der neuen Kante werden anhand der `result`-Spezifikation berechnet.

Für eine rein kontextfreie Grammatik genügen `lhs` und `rhs`, während die letzten drei Slots Information für den Featuremechanismus enthalten. Der Grammatiktyp kann die Regelstruktur um weitere Slots erweitern, wenn bei der Ausgabegenerierung zusätzliche regelabhängige Information verwendet werden soll.

`define-grammar` Funktion

`(define-grammar rules &key (ignore nil) start type fast)`

Diese Funktion erzeugt das Grammatikobjekt und gibt es zurück. Die Parameter haben folgende Bedeutung:

`rules` Liste der Regeln, in Listennotation. Das erste Element kann auch der Name eines Grammatiktyps sein.

`ignore` Liste der Kategorien, die im Parsebaum nicht erscheinen sollen.

`start` Startsymbol oder Liste von Startsymbolen.

`type` Name des Grammatiktyps der Grammatik.

`fast` Ist dieser Parameter ungleich `NIL`, dann nimmt der Grammatiklader weniger Überprüfungen vor und arbeitet dadurch schneller. Dies ist bei einer ausgetesteten Grammatik sinnvoll.

Die Funktion `define-grammar` bestimmt zuerst den Grammatiktyp. Ist keiner angegeben, wird nach einem Grammatiktyp gesucht, zu dem die gegebenen Regeln passen. Dann werden die gelesenen Regeln in Regelobjekte umgewandelt und die Stringkategorien bestimmt. Anschließend indiziert `define-grammar` die Regeln für den bottom-up- und den top-down-Modus und legt die entsprechenden Zugriffsclosures im Grammatikobjekt ab.

`load-grammar` Funktion

`(load-grammar grfile &key (ignore nil) start type fast)`

`grfile` ist ein Dateiname für die Regeldatei. In der Regeldatei kann als erstes Element der Name des Grammatiktyps stehen. In diesem Fall muß das Argument `type` nicht angegeben werden. Die Argumente `ignore`, `start` und `fast` entsprechen denen in `define-grammar`. Nachdem `load-grammar` die Datei gelesen hat, ruft es `define-grammar` mit dem Grammatiktyp und der Regelmenge auf.

Für den Grammatiklader sind bei der Definition eines Grammatiktyps folgende Optionen anzugeben:

`create-rule` Der Regelkonstruktor ist eine Funktion, die aus der Listennotation der Regel das Regelobjekt erzeugt.

Default: – , die Option muß angegeben werden.

`rule-test` Diese Prädikat überprüft, ob die Syntax der Regel in diesem Grammatiktyp zulässig ist.

Default: – , die Option muß angegeben werden.

Ein Beispiel für eine Grammatik steht in Anhang A.2.

5.3 Lexikon

Ein Lexikon definiert man, indem man eine Instanz der Struktur `lex` erzeugt. Ihre Slots enthalten einen Satz von Zugriffsfunktionen. Eine solche Lexikonstruktur kann dem Parser dann als Argument übergeben werden. Das Defaultlexikon befindet sich in der Variable `*lexicon*`.

Das Lexikon liefert zu einem Eingabeelement eine Liste von Einträgen zurück. Jeder Eintrag besteht aus Attribut-Wertpaaren. Die benötigten Angaben hängen vom Grammatiktyp ab.

`lex` Struktur

Die folgenden Komponenten der Struktur `lex` werden von allen Grammatiktypen benötigt.

`entries` Diese Funktion liefert für ein Wort der Eingabesequenz eine Liste von passenden Lexikoneinträgen. Der Aufbau der Lexikoneinträge ist beliebig. Die Funktion hat folgende Argumente:

`word` Wort, das im Lexikon nachgeschlagen werden soll

`ignore-cap` Beim ersten Wort der Eingabesequenz gibt man für diesen Parameter `t` an. In diesem Fall liefert die Groß- und Kleinschreibung keine Kategorieinformation und sollte vom Lexikon ignoriert werden.

`cat` Die Funktion bestimmt für einen Lexikoneintrag die Kategorie.

`word` Die Funktion bestimmt zu einem Lexikoneintrag die Grundform des Wortes.

Die restlichen Komponenten von `lex` müssen dagegen nur für bestimmte Grammatiktypen definiert sein.

`form` Die Funktion liefert zu einem Lexikoneintrag die ursprüngliche Wortform.
Default: entsprechend der Komponente `word`

`conf` Die Funktion gibt den Konfidenzfaktor des Eintrags zurück.
Default: 1

`attr` Die Funktion liefert für einen Eintrag die Featureliste.
Default: `nil`

`sem` Die Funktion gibt die semantische Information aus dem Eintrag zurück.
Default: `nil`

`make-entry` Diese Funktion erzeugt einen Lexikoneintrag.

`entry-p` Diese Prädikatfunktion gibt an, ob ein Objekt ein Lexikoneintrag ist.

Die beiden Slots `make-entry` und `entry-p` werden vom Parser nicht mehr benutzt und dienen nur noch zur Kompatibilität mit alten Lexika. Ein Beispiel für ein Lexikon findet sich in Anhang A.1.

5.4 Der Grammatiktyp `:cf`

Der Grammatiktyp `:cf` benützt keine Features, so daß bei den Regeln nur der Regelkopf und der Regelrumpf wichtig ist.
In der Listennotation des Grammatiktyps `:cf` sind die Regeln Listen, deren erstes Element den Regelkopf und deren restliche Elemente den Regelrumpf darstellen. Die Regel NP $\longrightarrow$ NP PP sieht in Listennotation folgendermaßen aus:

```
(NP NP PP)
```

Außer den gewöhnlichen (Symbol-)Kategorien sind in den Regeln auch Stringkategorien zugelassen. Das bedeutet, daß an der entsprechenden Stelle der Eingabesequenz genau dieses Wort vorkommen muß. Eine mit „und“ verknüpfte Aufzählung zweier PROPN beschreibt man mit NP $\longrightarrow$ PROPN **und** PROPN. In der Listennotation der Regel ist „und“ ein String, während die anderen Kategorien Symbole sind.

```
(NP PROPN "und" PROPN)
```

Der `:cf`-Grammatiktyp verwendet keine Attribute, so daß für die Lesartkanten der Basistyp genügt. Da dieser Grammatiktyp weder Features noch Attribute einsetzt, benötigt er aus dem Lexikon lediglich die Kategorie und das Wort selbst.

5.5 Der Grammatiktyp `:sf`

Der `:sf`-Grammatikformalismus arbeitet mit flachen Featurelisten, im Gegensatz zu den Unifikationsgrammatiken, die geschachtelte Ausdrücke verwenden. Ziel der Entwicklung von Grammatiken mit flachen Featurelisten ist es, einfache Tests über den morpho-syntaktischen Merkmalen in den Formalismus zu integrieren. Dabei wird insbesondere auf eine effiziente Verarbeitbarkeit der Strukturen Wert gelegt und bestimmte Schwächen in den Ausdrucksmöglichkeiten in Kauf genommen. Beispielsweise lassen sich keine Abhängigkeiten zwischen den

Werten verschiedener Features beschreiben. Diese Einschränkungen sind allerdings weniger gravierend, wenn man berücksichtigt, daß beim Parsen oft nach einer möglichst effizienten Abbildung von natürlicher Sprache in eine konzeptuelle Struktur gesucht wird und nicht nach einer möglichst exakten Beschreibung der Eigenschaften einer natürlichen Sprache. Ziel ist es also nicht, wohlgeformte von nicht wohlgeformten Eingabesequenzen unterscheiden zu können, sondern für möglichst viele der zu erwartenden Eingabesequenzen eine Ausgabestruktur zu erzeugen, die von nachfolgenden Komponenten verarbeitet werden kann.

Die Featureangabe des Grammatiktyps `:sf` ist eine Liste aus Featurenamen und deren Wertemengen in folgender Form:

⟨*fea-angabe*⟩ ::=
    ((⟨*fea-name*⟩⟨*fea-val*⟩)*)

⟨*fea-val*⟩ ::=
    ⟨*value*⟩
  | (**:or** ⟨*value*⟩+)

Beispiel: `((num pl)(pers (:or 1 3)))`

Wenn die Angabe für ein Feature fehlt, dann bedeutet das nicht etwa "leere Wertemenge" sondern "beliebiger Wert". Diese Definition erlaubt es, unvollständige Angaben auf einfache Art zu behandeln.

Außerdem gibt es beim Grammatiktyp `:sf` zwei Attribute:

- Der Konfidenzfaktor ist eine Zahl. Er gibt die Zuverlässigkeit der Informationen in einem Lexikoneintrag an und bewertet die Prioritäten von Grammatikregeln.
- Die Semantikspezifikation einer Grammatikregel beschreibt den Aufbau einer semantischen Struktur. So spezifiziert man Ausgaben für die Weiterverarbeitung.

Features und Attribute müssen an folgenden Stellen im Parser berücksichtigt werden:

- Im Lexikoneintrag, damit die Information für die Terminalsymbole zur Verfügung steht.
- Bei den Regeln, damit man die Propagierung der Information vom grammatikalischen Konstrukt abhängig machen kann.
- In den Nichtterminalkanten, weil der Grammatiktyp dort das Ergebnis von Ableitungen ablegt.

Der Rest des Abschnitts beschreibt die Behandlung der Information für die einzelnen Bereiche.

5.5.1 Datenstrukturen

Um die zusätzliche Information abzulegen, erweitert der Grammatiktyp `:sf` die Datenstrukturen für die Grammatikregeln und die Lesartkanten folgendermaßen:

`sf-rule` Struktur

Die Regel erhält zusätzlich zu den Slots aus `rule` folgende Slots für `:sf`-spezifische Information:

`conf` Konfidenzfaktor der Regel

`sem` Semantikspezifikation der Regel

`sf-edge` Struktur

Für die Attribute und Features erweitert der Kantentyp `sf-edge` den Basistyp `tree-edge` der Lesartkante um folgende Slots:

`result` Spezifikation für neue Feature-Liste. Der Wert wird aus dem Slot `result` der Regel entnommen.

`sem` Spezifikation semantischer Information. Sie wird von einem Satz von Makros verarbeitet.

`conf` Sicherheitsfaktor, eine Zahl zwischen 0 und 1. Ist kein Sicherheitsfaktor im Lexikon angegeben (`NIL`), wird 1 angenommen. Bei Regelanwendungen werden Sicherheitsfaktoren multipliziert.

5.5.2 Lexika für `:sf`-Grammatiken

Von der lexikalischen Analyse benötigt der Parser für jedes Eingabeelement eine oder mehrere Featurelisten für die syntaktische Analyse und die Attributwerte für den Ausgabenerator.
Für das Wort „Musikinstrumente“ sieht diese Information folgendermaßen aus:

```
(:LEX  "Musikinstrumente"
 :WORD "MUSIKINSTRUMENT"
 :COMP ("MUSIK" "INSTRUMENT" "E")
 :CAT  N
 :CONF 0.8
 :ATTR (GEN neut NUM (sg pl)
    KAS (nom gen dat akk))
 :SEM  "MUSIKINSTRUMENT")
```

Die Analysephasen des Parsers verwenden diese Information. Für die syntaktische Analyse sind die Kategorie und die mit `:attr` markierten Features wichtig. Der Konfidenzfaktor und die Semantikspezifikation werden für die Lesartgenerierung benutzt.

5.5.3 :sf-Grammatikregeln

Die Listennotation der Grammatikregel enthält außer dem Regelkopf und dem Regelrumpf einen Konfidenzfaktor, Featurebedingungen, Featurezuweisungen, die Semantikspezifikation und einen (bisher) unbenutzten Reserveslot.

Die folgende Grammatikregel besagt, daß eine Nominalphrase `NP` aus einem Artikel `DET` und einem Nomen `N` bestehen kann und daß Artikel und Nomen bezüglich Numerus, Genus und Kasus übereinstimmen müssen. Außerdem werden diese Feature-Werte an die generierte `NP` weitergegeben. Die Semantik der Nominalphrase `NP` ergibt sich einfach durch Übernahme der Semantik des Nomens:

```
(NP -> (DET N)
    1
    ((= (DET num) (N num))
     (= (DET kas) (N kas))
     (= (DET gen) (N gen)))
    ((num (DET num) (N num))
     (kas (DET kas) (N kas))
     (gen (DET gen) (N gen)))
    (&sem N)
    ())
```

Im folgenden werden die Bestandteile der Regeln und deren Einsatzmöglichkeiten im einzelnen beschrieben.

5.5.3.1 Kontextfreier Anteil

Die ersten drei Elemente einer solchen Regel beschreiben den kontextfreien Anteil.

- Regelkopf (Kategorie der linken Seite)
- Pfeil
- Regelrumpf (Kategorien der rechten Seite als Liste)

Falls auf der rechten Seite eine Kategorie mehrfach vorkommt und später eine eindeutige Referenz auf eines der Elemente benötigt wird, wird an die Kategorien ein Index mit einem Punkt angehängt.

In folgender Regel sind die NPs auf der rechten Seite mit einem Index versehen, um in den Tests und Zuweisungen eine eindeutige Referenz auf die Elemente der rechten Regelseite sicherzustellen:

```
(NP -> (NP.1 KONJ NP.2) ...)
```

5.5.3.2 Sicherheitsfaktor

Das vierte Element einer Grammatikregel besteht aus einem numerischen Wert, der die Konfidenz (Priorität, Sicherheitsfaktor) der Regel festlegt. Normalerweise ist sie auf 1 gesetzt. Durch einen höheren bzw. niedrigeren Wert kann das

Gewicht einer Regel erhöht oder verringert werden. Zusammen mit den von der lexikalischen Analyse gelieferten Sicherheitsfaktoren der Eingabeelemente dienen die Regelkonfidenzen dazu, die verschiedenen Lesarten gegeneinander zu gewichten und gegebenenfalls eines der Resultate auszuwählen. Wie die Sicherheitsfaktoren der lexikalischen Analyse werden die Regelkonfidenzen nach heuristischen Prinzipien vergeben – sie können nicht als Wahrscheinlichkeiten aufgefaßt werden.

Im aktuellen Ansatz werden die Sicherheitsfaktoren der Resultate der lexikalischen Analyse mit den Konfidenzfaktoren der Regeln multipliziert. Wird eine Regel r mit Sicherheitsfaktor s_r auf die Konstituenten $w_1 \ldots w_n$ mit den Sicherheitsfaktoren $s_{w_1} \ldots ...s_{w_n}$ angewendet, so wird der Sicherheitsfaktor des abgeleiteten Symbols u als $s_u = s_r * s_{w_1} * \ldots * s_{w_n}$ berechnet.

Eine solche Verarbeitung der Sicherheitsfaktoren und Regelprioritäten stellt sicher, daß Resultate mit höheren Konfidenzwerten aus der lexikalischen Analyse, sowie Resultate, in die Regeln mit höheren Prioritäten eingehen, bevorzugt werden. Der Sicherheitsfaktor ist ein Attribut und wird bei der Ausgabegenerierung berechnet.

Sicherheitsfaktoren werden bei der Berechnung entsprechend den Angaben in der globalen Variable `*numfix*` gerundet.

`*numfix*` Variable

Gibt die Anzahl der Nachkommastellen für die Rundung von Sicherheitsfaktoren an. Der Defaultwert ist 2.

5.5.3.3 Featuretests

Dem Konfidenzfaktor folgt eine Liste mit Tests über den morpho-syntaktischen Merkmalen. Der Parser wendet eine Regel nur an, wenn alle Featurebedingungen erfüllt sind. Mit Hilfe eines solchen Tests wird überprüft, ob die Werte bestimmter Features von Kategorien des Regelrumpfs zueinander passen. Da Features mehrere alternative Werte haben können, gilt ein Test als erfüllt, wenn die Schnittmenge der jeweiligen Wertemengen nicht leer ist, d.h. die beiden Features wenigstens einen gemeinsamen Wert haben. Auch wenn die Bedingung mit einem `=` notiert wird, sollte man deswegen anstatt von "Gleichheit" eher von "Kompatibilität" der Featurewerte sprechen.

Außerdem ist ein Test erfüllt, wenn eine der beteiligten Wertemengen `nil` ist, d.h. über das betreffende Feature keine Information verfügbar ist. Oft sind Featureinformationen während der Entwicklung einer Grammatik noch unvollständig. Die Featurebedingungen verbieten daher den Ableitungsschritt nur bei widersprüchlichen Featurewerten und nicht, wenn eine Angabe fehlt. Damit kann man eine Grammatik grob entwerfen, sie testen und ihre Wirkung dann durch Featurebedingungen schrittweise verbessern.

Bei der Kantenverschmelzung sind die Bedingungen für die Features dagegen strenger. Damit zwei Kanten verschmolzen werden, müssen ihre Features wirklich äquivalent sein, d.h. die gleichen Wertemengen besitzen. (vgl. Abschnitt 4.3.2.5).

Wie gesagt bedeutet bei den Featuretests "=" soviel wie "ist kompatibel mit", d.h. die jeweiligen Schnittmengen sind nicht leer. Außer der Kompatibilität verschiedener Featurewerte des Regelrumpfs kann ein Feature auch auf Kompatibilität mit einem fest vorgegebenen Wert oder einer fest vorgegebenen Menge alternativer Werte getestet werden. Damit gibt es folgende Möglichkeiten:

- `(= (<cat1> <f1>) (<cat2> <f2>))`
  Der Wert des Features `<f1>` der Kante `<cat1>` muß zum Wert des Features `<f2>` der Kante `<cat2>` kompatibel sein. Sowohl `<cat1>` und `<cat2>` als auch `<f1>` und `<f2>` können voneinander verschieden sein, müssen aber nicht.

  Die folgende Bedingung ist erfüllt, wenn Artikel und Nomen wenigstens in einem Wert des Numerus-Features `num` übereinstimmen oder einer der beiden Featurewerte unbestimmt ist.

  ```
  (= (DET num) (N num))
  ```

- `(= (<cat> <f>) <atom>)`

  `<atom>` muß als Wert von `<f>` in `<cat>` vorkommen.

  Die Bedingung in diesem Beispiel fordert, daß die Nominalphrase NP im Singular steht (oder das Feature unbestimmt ist).

  ```
  (= (NP num) sg)
  ```

- `(= (<cat> <f>) (:or <atom> <atom> ...))`

  Mindestens eines der Atome muß als Wert des Features `<f>` in `<cat>` auftreten.

  Wenn die Nominalphrase NP in der dritten oder der ersten Person stehen soll, lautet die entsprechende Featurebedingung folgendermaßen:

  ```
  (= (NP pers) (:or 1 3))
  ```

5.5.3.4 Featurezuweisungen

Das sechste Element einer Grammatikregel besteht aus einer Liste von Featurezuweisungen für den Regelkopf. Das erste Element einer Zuweisung gibt dabei jeweils das Feature an, an das die Zuweisung erfolgt, der Rest die Wertangabe. Als Wertangaben sind ein konstanter Wert, eine konstante Wertemenge, die Wertemenge eines Features des Regelrumpfs und Schnittmengen solcher Angaben zulässig.
Welche Features dem Regelkopf zugewiesen werden sollen, wird durch eine Anweisungen der Form `(<f1> <val>+)` spezifiziert. Dabei wird dem Feature des Regelkopfs mit Namen `<f1>` der Wert `<val>+` zugewiesen.

Die o.g. Fälle von Wertangaben werden folgendermaßen notiert:

- `<atom>` Der neue Wert ist `<atom>` (genauer: die Wertemenge ist die Menge, die nur `<atom>` enthält)

  Der Ausdruck `(pers 3)` setzt z.B. das Feature `pers` des Regelkopfs auf den Wert `3`.

- `(:or <atom> <atom> ...)`

  Die neue Wertemenge des Features ist die Menge der angegebenen Atome.

  Mit `(pers (:or 1 3))` weist man dem Feature `pers` die Werte 1 und 3 zu.

- `(<cat> <f2>)`

  Die neue Wertemenge wird aus dem Feature `<f2>` der Kategorie `<cat>` entnommen. `<cat>` ist dabei eine Kategorie aus dem Regelrumpf.

  Der Ausdruck `(num (N num))` übernimmt z.B. den Wert des Features `num` für das neue Symbol vom gleichen Feature der Konstituente mit der Kategorie `N`.

- `(<f1> <val> <val> ...)`

  Wenn mehrere Werte angegeben werden, belegt der Grammatiktyp das Feature `<f1>` des Regelkopfs mit der *Schnittmenge* der angegebenen Werte.

  Der Ausdruck `(pers (N pers) (:or 1 3))` weist dem Feature `pers` die Wertemenge desselben Features der Konstituente `N` geschnitten mit der Menge $\{1, 3\}$ zu.

Die Trennung von Bedingungs- und Zuweisungsteil in den Grammatikregeln hat zur Folge, daß nicht alle Informationen, die auf einer Ebene der syntaktischen Struktur zur Verfügung stehen, automatisch auch auf der übergeordneten Ebene verfügbar sind. Alle Features, die auf einer höheren Ebene benötigt werden, müssen also explizit im Zuweisungsteil einer Regel übergeben werden. Zusätzlich können auf jeder Ebene neue Features und Featurewerte eingeführt werden.

5.5.3.5 Semantikspezifikation

Schließlich enthält jede Grammatikregel noch eine Semantikspezifikation in Form eines beliebigen Lisp-Ausdrucks, der dem Regelkopf eine Semantik zuordnet. Die Semantikspezifikation wird bei der Ausgabegenerierung evaluiert, wobei die Lesartkante in der `special` deklarierten Kantenvariablen `edge` im Ausdruck zur Verfügung steht. In einer solchen Semantikspezifikation können zwei vordefinierte Makros verwendet werden, um auf Semantikresultate von Kategorien des Regelrumpfs oder auf Featurewerte zuzugreifen:

- `(&sem <rhs-cat>)`

  Evaluiert zum Resultat der semantischen Analyse für die Kategorie `<rhs-cat>` des Regelrumpfs.

- `(&attr <fea-name>)`

  Evaluiert zum Wert des Features `<fea-name>` des Regelkopfs

- `(&attr (<rhs-cat> <fea-name>))`

  Evaluiert zum Wert des Features `<fea-name>` für die Kategorie `<rhs-cat>` des Regelrumpfs.

Diese Aufrufe kann man in beliebigen Lispcode einbetten, z.B. fügt der Ausdruck `(cons (&sem ADJ) (&sem N))` die semantische Information von `ADJ` als erstes Element zur semantischen Information von `N` hinzu und gibt diese Struktur zurück.
In einer Semantikspezifikation kann man auf Featurewerte zugreifen. Der Bedingungsteil einer Regel hat dagegen *keinen* Zugriff auf Teilresultate der semantischen Analyse. Der Grund ist, daß die Features in der syntaktischen Phase erzeugt und benutzt werden, während die Semantikspezifikation erst als Attribut bei der Ausgabegenerierung berechnet wird.

5.5.4 Syntax von Grammatikregeln

Die Syntax von Grammatikregeln mit flachen Featurelisten ist im folgenden zusammengefaßt:

⟨*rule*⟩ ::=
  (⟨*lhs-cat*⟩ `->` (⟨*rhs-cat*⟩+)
  ⟨*confidence*⟩
  ⟨*tests*⟩
  ⟨*actions*⟩
  ⟨*semantics*⟩
  ⟨*spare*⟩)

⟨*lhs-cat*⟩ ::=
  ⟨*symbol*⟩

⟨*rhs-cat*⟩ ::=
  ⟨*symbol*⟩
  | ⟨*symbol*⟩ `.` ⟨*index*⟩

⟨*index*⟩ ::=
  ⟨*integer*⟩

⟨*confidence*⟩ ::=
  ⟨*number*⟩

⟨*tests*⟩ ::=
((= ⟨*fea-arg*⟩ ⟨*fea-arg*⟩)*)

⟨*fea-arg*⟩ ::=
(⟨*rhs-cat*⟩ ⟨*fea-name*⟩)
| ⟨*value*⟩
| (**:or** ⟨*value*⟩+)

⟨*actions*⟩ ::=
((⟨*fea-name*⟩ ⟨*fea-arg*⟩+)*)

⟨*fea-name*⟩ ::=
⟨*symbol*⟩

⟨*value*⟩ ::=
⟨*atom*⟩

⟨*semantics*⟩ ::=
⟨*s-expression*⟩

⟨*spare*⟩ ::=
⟨*s-expression*⟩

Im Lispausdruck ⟨s-expression⟩ der Semantikspezifikation sind folgende Aufrufe zulässig:

⟨*s-expression*⟩ ::=
(**&sem** ⟨*rhs-cat*⟩)
| (**&attr** ⟨*fea-name*⟩)
| (**&attr** (⟨*rhs-cat*⟩ ⟨*fea-name*⟩))

Kapitel 6

Referenzhandbuch

6.1 Parserfunktionen

ChaPLin stellt eine Reihe von Parserfunktionen zur Verfügung, die die einzelnen Phasen des Algorithmus 2 (S. 12) abdecken. In Phase 1 zerlegt der Scanner einen Eingabestring in Eingabeelemente. Phase 2 umfaßt die lexikalische Analyse und den Chartaufbau; sie erzeugt also die Terminalsymbole. Die eigentliche syntaktische Analyse ist dann Phase 3 und die nachfolgende Ausgabegenerierung Phase 4. Abbildung 3.1 stellt die beteiligten Komponenten von ChaPLin dar. Tabelle 6.1 zeigt, welche Parserfunktionen welche Analysephasen abdecken.

6.1.1 Argumente der Parserfunktionen

In den verschiedenen Phasen werden unterschiedliche Eingabedaten verarbeitet. Phase 1 erhält einen String `string` als Eingabe. Die Funktionen `scan-line` und `parse-line` lesen diesen String vom Terminal. Phase 2 erhält ein Eingabeelement `item` oder eine Eingabefolge `seq` als Argument. Die folgenden Phasen finden die benötigte Information in der Chart.

Die Optionen werden durch Keywordargumente spezifiziert und sind bei allen Parserfunktionen gleich. Tabelle 6.1 gibt an, welche Optionen die einzelnen Funktionen tatsächlich berücksichtigen.

Funktion	Phasen				Eingabe	g	m	l	f	i	o
`parse-line`	1	2	3	4	–	+	+	+	+	-	+
`parse`		2	3	4	`seq`	+	+	+	+	-	+
`parse-next`		2	3	4	`item`	+	-	+	+	+	+
`scan-line`	1				–	-	-	-	-	-	-
`scan`	1				`string`	-	-	-	-	-	-
`build-chart`		2			`seq`	+	+	+	-	-	-
`parse-rest`			3	4	–	+	+	-	+	-	+
`build-tree`				4	–	+	+	-	+	-	+

Tabelle 6.1: Tabelle der Parserfunktionen

`grammar` **(g)** Gibt die Grammatik an. Wird keine Grammatik angegeben, dient bei der Initialisierung der Chart `*grammar*` als Defaultwert. Anschliessend wird die Grammatik in `*parse-grammar*` abgelegt und bis zur nächsten Initialisierung der Chart als Defaultwert für `grammar` benutzt.

`mode` **(m)** `:bu` für bottom up, `:td` für top down.

`lexicon` **(l)** Das Lexikon ist eine Instanz der Struktur `lex`.
Defaultwert ist `*lexicon*`.

`find` **(f)** Startkategorie oder Liste von Startkategorien. Eine Kategorie ist ein Symbol. Der Defaultwert ist die Startkategorie der Grammatik. Hat `find` den Wert `NIL`, dann sind alle Symbole als Startkategorie zugelassen. Im Top-down-Modus darf nur ein Startsymbol angegeben werden.

`igncap` **(i)** Boolscher Wert, Default ist `NIL`. Dieses Argument gibt an, ob die Groß- und Kleinschreibung von der lexikalischen Analyse ignoriert werden soll. Beim wortweise inkrementellen Parsen sollte `igncap` zu Beginn des Satzes auf `t` gesetzt werden, da dort die Großschreibung keine lexikalische Information liefert. Für den Rest des Satzes ist der Defaultwert `NIL` angemessen.

`output` **(o)** Ausgabespezifikation, Default ist `*output*`
Die Ausgabespezifikation ist entweder ein Keyword als Kurzspezifikation oder – für eine präzisere Kontrolle – eine Liste von Keywords und Werten. Die Funktionsweise des Ausgabegenerators wird im Abschnitt 4.4 erklärt.

6.1.2 Beschreibung der Parserfunktionen

Die Schnittstellenfunktionen akzeptieren beliebige Keywordparameter. Im Kopf sind nur die Parameter angegeben, die von der Funktion wirklich benutzt werden.

`build-chart` Funktion
`(build-chart inputseq &key mode lexicon grammar)`
Die Funktion `build-chart` initialisiert die Chart mit der Eingabesequenz `inputseq`. Dafür wird das Lexikon `lexicon` benötigt. Beim bottom-up-Parsen wird die Chart automatisch mit der angegebenen Grammatik aktiviert, beim top-down-Parsen nicht. Eine leere Chart zum wortweise inkrementellen Parsen erzeugt man mit `(build-chart ())`.

`build-tree` Funktion
`(build-tree &key output grammar lexicon find)`
`build-tree` baut die Ausgabestruktur aus der Chart auf. Wichtig ist hier vor allem das Startsymbol `find` und die Ausgabespezifikation `output`.

`loop-parse-incremental` Funktion

`(loop-parse-incremental &key output lexicon grammar find edge-count)`

Aufruf von `parse-incremental` in einer Schleife, die mit einer leeren Eingabe beendet wird. Argumente siehe `parse-incremental`.

`parse` Funktion

`(parse inputseq &key output mode grammar lexicon find)`

Die Funktion `parse` analysiert eine Sequenz von Eingabeelementen oder Lexikoneinträgen und generiert eine Ausgabestruktur.

`parse-and-show-time` Funktion

`(parse-and-show-time inputseq &key output mode grammar lexicon find)`

Die Funktion arbeitet wie `parse` gibt aber zusätzlich die zum Parsen benötigte Zeit aus. Für den Parameter `:mode` ist hier auch `:both` erlaubt. Dann mißt die Funktion den Zeitbedarf für beide Modi.

`parse-incremental` Funktion

`(parse-incremental &key output lexicon grammar find)`

`parse-incremental` liest einen Satz vom Terminal und parst bereits während der Eingabe. Es wird ein spezieller Scanner verwendet, der das Löschen von Zeichen behandeln kann. Sein Verhalten kann daher vom Verhalten des üblichen ATN-Scanners abweichen.

Hat der inkrementelle Scanner ein Eingabeelement vollständig erkannt, dann ruft er `parse-next` auf. Während `parse-next` arbeitet, verarbeitet der inkrementelle Scanner keine weiteren Tastendrucke. Wenn der Benutzer so viele Zeichen löscht, daß ein bereits weitergegebenes Eingabeelement zurückgenommen werden muß, dann wird der ganze noch gültige Rest des Satzes erneut geparst.

Das genaue Verhalten hängt vom Lispsystem und dem verwendeten Terminal ab. Wenn `parse-next` schnell genug ist oder das Terminal die Eingabe puffert, nimmt der Benutzer den Zeitbedarf von `parse-next` nicht wahr. Das zeichenweise inkrementelle Parsen ist unter Allegro Common Lisp 4.2 auf SUN-SPARC und unter MCL auf Apple Macintosh getestet.

`parse-line` Funktion

`(parse-line &key output mode grammar lexicon find)`

Die Funktion liest eine Zeile vom Terminal, zerlegt sie mit `scan` in Eingabeelemente und ruft `parse` auf.

`parse-line-and-display-tree` Funktion

`(parse-line-and-display-tree &key output mode grammar ...)`

Kombination aus `parse-line` und `parse-and-display-tree`.

`parse-line-and-show-time` Funktion

`(parse-line-and-show-time &key output mode grammar lexicon find)`

Kombination aus `parse-line` und `parse-and-show-time`.

`parse-next` Funktion

`(parse-next element &key grammar lexicon find ignore-cap output)`

Die Funktion `parse-next` parst wortweise inkrementell. Sie verlängert die Chart um das Eingabeelement `element` und parst den Satz im bottom-up-Modus.

`parse-rest` Funktion

`(parse-rest &key grammar mode find output)`

`parse-rest` parst mit den bereits in der Chart vorhandenen Daten. Daher wird auch kein Satz oder Eingabeelement angegeben. Anstelle der Funktion `parse` kann man auch `build-chart` und anschließend `parse-rest` aufrufen.

Es ist möglich, die Analyse einer bestehenden Chart mit einer anderen Grammatik fortzusetzen. Gibt man im Argument `grammar` eine andere Grammatik an, als bisher verwendet wurde, dann reaktiviert die Funktion `parse-rest` zuerst die Chart mit der neuen Grammatik.

6.1.3 Globale Defaults

Die Defaultwerte für die Keywordargumente der Parserfunktionen sind in globalen Variablen festgelegt. Für einige dieser Variablen gibt es wiederum Defaultwerte beim Laden des Parsers.

`*find*` Variable

Default für das Startsymbol.

`*grammar*` Variable

Das Symbol `*grammar*` wird aus dem in `*nlpkg*` spezifizierten Package importiert. Diese Variable enthält die Defaultgrammatik. Wenn man einen neuen Satz analysiert, d.h. die Chart initialisiert, dann wird diese Grammatik benutzt. Für diese Variable gibt es keinen Defaultwert, sie wird sinnvollerweise beim Laden einer Grammatik belegt.

`*lexicon*` Variable

Das Symbol `*lexicon*` wird aus dem in `*nlpkg*` spezifizierten Package importiert. Diese Variable enthält das Defaultlexikon. Für diese Variable gibt es keinen Defaultwert, sie wird sinnvollerweise beim Laden eines Lexikons belegt.

`*mode*` Variable

Default für den Parsemodus. Defaultwert beim Laden ist `:bu` für bottom-up.

`*output*` Variable

Diese Variable enthält den Default für die Ausgabespezifikation. Default ist `:tree`, d.h. es werden Parsebäume erzeugt.

`*parse-grammar*` Variable
Im Normalfall möchte man für alle Phasen des Parseprozesses dieselbe Grammatik verwenden. In dieser Variablen wird daher die aktuelle Grammatik abgelegt. Die Variable `*parse-grammar*` ist der Defaultwert, wenn man eine Funktion aufruft, die mit einer bestehenden Chart weiterarbeitet. Funktionen, die eine neue Chart anlegen, belegen `*parse-grammar*` mit dem Wert des Parameters `grammar`.

`*tree-default*` Variable
Default für die Baumspezifikation, die das Format der Parsebaumknoten bestimmt. Default beim Laden ist:

```
'(:node    edge-cat
  :lex     (:cat :lex)
  :struct  (:cat . :contents)))
```

6.1.4 Ausgabe

Die Ausgabegenerierung verläuft nach Algorithmus 3 (S. 34). Den Ausgabegenerator steuert man mit der Ausgabespezifikation in Form eines einzelnen Schlüsselworts (Kurzspezifikation) oder einer Liste aus Schlüsselwörtern und Werten.

Wenn eine Kurzspezifikation angegeben wird, expandiert die `build-fn`-Funktion des Grammatiktyps das Kürzel zu einer Liste. Bei der Ausgabeform `:tree` setzt die Funktion dabei die Defaults aus `*tree-default*` für die Baumspezifikation ein.

In den Grammatiktypen `:sf` und `:cf` sind derzeit folgende Kurzspezifikationen definiert:

`nil` Siehe `:succ`.

`:succ` Liste der Kategorien der erfolgreichen Kanten

`:tree` Syntaxbaum (Default)

`:alt` Alternativen im Syntaxbaum

`:frag` Fragmente, falls keine vollständige Analyse

`:num` Syntaxbäume mit Sicherheitsfaktor (nur beim `:sf`-Grammatiktyp)

`:allnum` Sicherheitsfaktor an allen Knoten (nur beim `:sf`-Grammatiktyp)

`:sem` Aufbau einer Struktur aus dem Semantik Slot (nur beim `:sf`-Grammatiktyp)

Die Hauptausgabefunktion `create-output` erhält die Ausgabespezifikation als Liste aus Schlüsselwörtern und Werten. Sie wird entweder direkt angegeben oder entsteht durch die o.g. Expansion einer Kurzspezifikation. Algorithmus 7 beschreibt, welche Ausgabeformen den Angaben in der Ausgabespezifikation zugeordnet sind.

Algorithmus 7
Die Aktion wird ausgeführt, wenn der Wert des Schlüsselworts nicht `NIL` ist.

`:result` Gib den Spezifikationswert zurück. (Das Ergebnis wurde bereits von Übersetzungsfunktion `build-fn` berechnet.)

`:last` Nimm die eingehenden Kanten des letzten Knotens als Kandidaten für erfolgreiche Kanten. (Im Normalfall gehen erfolgreiche Kanten vom ersten zum letzten Knoten der Chart.)

`:cat` Gib eine Liste der Kategorien der erfolgreichen Kanten zurück.

`:succ` Wenn erfolgreiche Kanten existieren, gib `t` zurück, sonst `nil`.

`:count` Gib Anzahl der Lesarten zurück.

`:tree` Erzeuge Parsebäume für alle erfolgreichen Kanten.

`:frag` Wenn es keine erfolgreichen Kanten gibt, suche nach Fragmentfolgen (vgl. Funktion `fragment-report`).

Die Ausgabeformen `:tree` und `:frag` erzeugen Lesartkanten und Parsebäume, was verhältnismäßig aufwendig ist. Die anderen Ausgabeformen lassen sich aus der Chart dagegen sehr schnell ablesen.

In der Phase der Lesarterzeugung werden Lesartkanten erzeugt, die die Attribute enthalten. Die Lesarten für eine syntaktische Kante werden noch gefiltert, d.h. mit Hilfe der Attributinformation kann eine Auswahl der Lesarten vorgenommen werden. Der Lesartfilter erhält die Spezifikation als Argument. Beim Grammatiktyp `:sf` erkennt der Filter folgende Spezifikationswerte:

`:sort` Wenn der Wert `t` ist, werden die Kanten nach dem Wert des Sicherheitsfaktors absteigend sortiert.

`:nres` Der Wert ist eine Zahl. Es werden höchstens soviele Lesarten auf jeder Ebene eingetragen, alle anderen werden weggeworfen.

Möchte man nur die n besten Lesarten verwenden, dann sortiert man und gibt eine Anzahlbeschränkung an (`:sort t :nres` n). Wenn die Filterfunktion die Lesartliste nicht sortiert, dann ist die Reihenfolge der Lesarten undefiniert. Eine Anzahlbeschränkung ohne Sortierung wählt die Lesarten willkürlich aus.

Bei der Baumerzeugung werden zu den Lesartkanten Syntaxbäume aufgebaut. Die Baumspezifikation zur Steuerung der Knotenform enthält Angaben zu den drei Schlüsselwörtern `:lex`, `:struct` und `:node`. Die so angegebenen Spezifikationen sind Muster für die Struktur des Baumknoten, wobei jeder Bestandteil des Musters entsprechend Algorithmus 6 (S. 38) durch seinen Wert ersetzt wird. Die benötigte Information wird aus der Lesartkante entnommen.

Bei den Terminalsymbolen beschreiben `:lex`-Spezifikationen die Gestalt von Parsebaumblättern für Lexikoneinträge und `:struct`-Spezifikationen den Einbau von Unterparsebäumen für Struktureinträge.

Bei Nichtterminalsymbolen besteht der Parsebaum aus einem Knoten und der Liste der Unterbäume, wobei die Gestalt des Knotens durch die `:node`-Spezifikation gesteuert wird.

Am Ende der Ausgabegenerierung werden die Bäume auf oberster Ebene nachbearbeitet. Damit kann man der Wurzel des Parsebaums eine andere Form geben als den inneren Knoten. Die entsprechende Funktion des Grammatiktyps `grammar-type-tree-postproc` erhält ebenfalls die Ausgabespezifikation als Eingabe. Es sind derzeit zwei Nachbearbeitungsvarianten implementiert:

`:lex&struct` Erzeugt für ein Nichtterminalsymbol einen Struktureintrag, der als Eingabeelement für eine weitere Parsephase verwendet werden kann. So erzeugt man also die Struktureinträge, die man mit der `:struct`-Spezifikation wieder in einen Parsebaum einbauen kann. Für ein Terminalsymbol wird ein `:lex`-Eintrag erzeugt.

`:num` Fügt der Wurzel jedes Parsebaums den Sicherheitsfaktor hinzu.

6.2 Untersuchung und Analyse

Alle Ausgabefunktionen erhalten als Keywordargument `stream` den Ausgabestream. Default ist `t`, d.h. das Terminal.

6.2.1 Quantitative Untersuchungen

In diesem Abschnitt werden Funktionen vorgestellt, die einige interessante quantitative Analysen der Chart vornehmen. Quantitative Analysen sind eine wichtige Hilfe bei Effizienzproblemen oder bei der Entwicklung von Grammatiken. Bei einer quantitativen Analyse werden aber Einzeleffekte leicht von anderen Effekten überlagert und verfälscht, so daß diese Analysen nur einen Hinweis oder eine Tendenz angeben können. Quantitative Analysen zeigen, ob eine Grammatik zu restriktiv ist, so daß zuwenig Lesarten gefunden werden, oder zu großzügig, so daß sie eine zu hohe Mehrdeutigkeit aufweist.

`chart-analysis` Funktion

`(chart-analysis &key (stream t))`

Diese Funktion ruft die einzelnen Analysefunktionen mit geeigneten Defaultwerten auf. Die Teilanalysen werden im folgenden beschrieben.

`size-report` Funktion

`(size-report &key (stream t))`

Die Untersuchung gibt die Anzahl der Knoten aus und schlüsselt die Anzahl der Kanten nach den verschiedenen Kantentypen auf. Die dort aufgeführten *stillgelegten* Kanten entstehen, wenn auf der gleichen Chart mehrmals eine Ausgabe erzeugt wird und dafür die Lesartkanten neu berechnet werden.

Die folgenden Untersuchungen betrachten nur noch die (syntaktischen) inaktiven Kanten.

`parse-tree-report` Funktion

`(parse-tree-report &key (stream t)(succ (success-edges *parse-grammar*)))`

Diese Funktion untersucht den Parsewald und gibt die Anzahl der erfolgreichen (syntaktischen) Kanten und deren Lesarten aus. Außerdem wird bestimmt, wieviele Kanten wie oft im Parsewald enthalten sind.

Eine hohe Zahl unbenutzter Kanten weist darauf hin, daß der Parser viele Ableitungen im Bereich der kürzeren Konstrukte findet, die nachher nicht zu einem Satz vervollständigt werden können.

Hohe Benutzungsquoten für einzelne Kanten entstehen, wenn die Grammatik bei den langen Konstrukten Mehrdeutigkeiten enthält.

`fragment-report` Funktion

`(fragment-report &key (stream t))`

Diese Untersuchung ist sinnvoll, wenn der Satz nicht erfolgreich geparst wurde. In diesem Fall möchte man wissen, welche und wie große Teilstücke in der Chart erfolgreich geparst werden konnten. Betrachtet man eine unvollständige Chart, versucht man, darin möglichst große geparste, d.h. von einer Kante überspannte, Abschnitte – *Fragmente* – zu finden und den Satz als Folge solcher Fragmente darzustellen.

Bei der Suche nach Fragmentfolgen kann man zwei Teile der Chart (und damit des Satzes) völlig unabhängig voneinander betrachten, wenn die Chart durch Herausnehmen eines trennenden Knotens unzusammenhängend wird. Im folgenden steht – graphentheoretisch betrachtet – der Begriff *Komponente* für die Komponenten der Chart bezüglich 2-fachem Knotenzusammenhang. Ein Knoten trennt zwei Komponenten, wenn er von keiner Kante überspannt wird, d.h. wenn es keine Kante gibt, deren Anfangsknoten echt kleiner als der Trennknoten und deren Endknoten echt größer als der Trennknoten ist. Als erstes bestimmt die Fragmentanalyse die Zahl dieser Komponenten. Nach dem Eintragen der Terminalsymbole hat eine Chart soviele Komponenten wie Eingabeelemente; nach erfolgreichem Parsen dagegen eine Komponente. Der Vergleich zwischen der Anzahl der Komponenten und der Anzahl der Eingabeelemente ist also ein Maß für den Grad der Erfolglosigkeit des Parsings.

Die vollständig geparsten Abschnitte der Chart sind die *Grundbereiche*. Eine Menge von aufeinanderfolgenden Knoten heißt Grundbereich, wenn folgende Bedingungen gelten:

1. Grundbereiche werden von einer Kante überspannt, d.h. die Kante verbindet den ersten Knoten des Grundbereichs mit dem letzten. Damit sind Grundbereiche immer Untermengen von Komponenten.

2. Keine Kante aus dem Inneren des Bereichs (Bereich *ohne* Anfangs- und Endknoten) verlässt den Grundbereich, d.h. es gibt keine Kante, deren Anfangsknoten im Inneren des Grundbereichs liegt und deren Endknoten außerhalb des Bereichs liegt.

3. Keine Kante von außerhalb betritt den Grundbereich, d.h. es gibt keine

Kante, deren Anfangsknoten außerhalb des Grundbereichs liegt und deren Endknoten im Inneren liegt.

4. Der Grundbereich ist maximal, d.h. keine echte Obermenge des Grundbereichs erfüllt die Bedingungen 1.-3.

Möchte man das Ergebnis des partiellen Parsens als Folge von Chartfragmenten ausgeben, dann betrachtet man die Grundbereiche als unteilbare Einheiten. Jedes Fragment läßt sich als Vereinigung von Grundbereichen darstellen.

Fragmentfolgen durch die Chart werden bei der Ausgabeform `:frag` als Folge von Komponentenpfaden dargestellt. Jeder Komponentenpfad ist eine Liste von Pfaden von Fragmenten vom Anfang zum Ende der Komponente. Für jedes Fragment wird die Liste aller Parsebäume angegeben.

Wenn alle Komponenten der Chart gleichzeitig Grundbereiche sind, dann hat der Parser den Satz in eine Folge von vollständig erkannten, voneinander unabhängigen Abschnitte zerlegt. Diese Komponenten heißen *trivial*, denn für diesen Satz kann man auf einfache Art eine Folge von jeweils vollständig erkannten, voneinander unabhängigen Teilstrukturen konstruieren.

Nichttriviale Komponenten – die also nicht gleichzeitig Grundbereiche sind – verursachen bei der Suche nach Fragmentfolgen einen erhöhten Aufwand. Z.B. hat eine Chart, die außer den Terminalkanten nur eine Kante von Knoten 1 nach 3 und eine weitere von 2 nach 4 enthält, eine nichttriviale Komponente zwischen den Knoten 1 und 4.

`alt-report` Funktion

`(alt-report &key (stream t))`

`alt-report` untersucht die Häufigkeit von Kantenverschmelzungen. Er gibt an, wieviele Kanten jeweils wieviele verschiedene Ableitungen enthalten. Bei stark mehrdeutigen Grammatiken gibt es eine hohe Zahl von Kantenverschmelzungen.

`rule-length-report` Funktion

`(rule-length-report &key (stream t))`

Untersuchung über Verteilung der angewendeten Regellängen. Eine große Zahl von Regeln der Länge 1 deutet darauf hin, daß viele Regeln, die nur umbenennen, angewendet wurden.

Anschließend wird die Anzahl der Terminalkanten angegeben. Da die lexikalische Analyse für eine Wortform mehrere Kategorien finden kann, ist die Anzahl der Terminalkanten möglicherweise größer als die Anzahl der Eingabeelemente. Das Verhältnis von Terminalkantenzahl zur Anzahl der Eingabeelemente ist ein Hinweis auf Mehrdeutigkeiten in der lexikalischen Analyse.

`multiedge-report` Funktion

`(multiedge-report &key (stream t) (spec <spec>))`

Diese Funktion liefert Hinweise auf Mehrdeutigkeiten in der Analyse und auf die Wirksamkeit der Grammatikregeln und Featuremechanismen, indem sie untersucht, in welchen Maße die Chart Multikanten enthält.

Im Parameter `spec` kann eine Liste von Spezifikationen angegeben werden. Jede Spezifikation ist eine Liste aus einer Titelzeile für die Untersuchung und einem Vergleichsprädikat für Kanten. Der Bericht gibt die Stärken der Äquivalenzklassen von Kanten bezüglich dieses Prädikats an.
Die Defaultspezifikation enthält zwei Untersuchungen:

- Bei der Untersuchung "Allgemeine Multikanten" sind zwei Kanten äquivalent, wenn sie die gleichen Anfangs- und Endknoten besitzen. Eine Multikante entsteht durch Mehrdeutigkeiten in der Grammatik, denn der Parser leitet für einen Teil der Chart mehrere Symbole ab. Da Kantenverschmelzung verwendet wird, handelt es sich bei diesen Mehrdeutigkeiten wirklich um verschiedene Symbole, nicht nur um verschiedene Ableitungswege zum gleichen Symbol.
- Multikanten mit gleicher Kategorie unterscheiden sich nur durch ihre Features. Sie sind ein Hinweis darauf, daß Kanten mit verschiedenen Features und der selben Kategorie erzeugt werden.

Die Multikantenzahlen sind gering, wenn entweder die Grammatik auf der Chart beinahe eindeutige Lösungen findet, oder Regeln, die dasselbe Symbol ableiten, zu einer optimalen Wirkung der Kantenverschmelzung führen. Diese Tatsache zeigt auch, daß ChaPLin mit beiden Formen von Grammatiken gut arbeiten kann.

6.2.2 Ausgabe von Datenstrukturen

`describe-chart` Funktion
`(describe-chart )`
Gibt ein `cons` aus der größten gültigen Knotennummer und der größten gültigen Kantennummer zurück. Die Knotennumerierung beginnt mit Knoten `*left-vertex*` = 1, die Kantennumerierung mit Kante 0.

`display-agenda` Funktion
`(display-agenda &key (stream t))`
Die Agenda ist ein Stapel von Konfigurationen, die aus einer aktiven und einer inaktiven Kante bestehen. Sie wird von oben nach unten in folgendem Format ausgegeben:

```
<links>---<akt. Kat, Nr.>---<mitte>---<inact.Kat.Nr.>---<rechts>
```

`display-chart` Funktion
`(display-chart &key from edges cat to stream)`
Die Funktion gibt die Kanten der Chart aus und erhält folgende Argumente:

`from` Wird eine Zahl angegeben, werden nur Kanten angezeigt, die von dem Knoten mit der Nummer `from` ausgehen. Der Default ist `NIL`, dann zeigt `display-chart` Kanten mit beliebigem Anfangsknoten an.

`to` Entsprechend für den Endknoten der Kanten.

`edges` Diese Option bestimmt den auszugebenden Kantentyp. Folgende Schlüsselwörter sind zulässig:

`:inactive` Nur inaktive Kanten ausgeben (Default)

`:active` Nur aktive Kanten ausgeben

`:tree` Nur Lesartkanten ausgeben.

`:all` Kanten aller drei Typen ausgeben.

`cat display-chart` gibt nur Kanten mit der hier angegebenen Kategorie aus. Default ist `nil`, dann werden Kanten mit beliebiger Kategorie ausgegeben.

`display-edge` Funktion

`(display-edge (e &key (stream t)))`

Die Printfunktion der Struktur `edge` gibt nur eine einzeilige Kurzinformation aus, die Funktion `display-edge` zusätzlich noch den Inhalt (bei inaktiven Kanten alle Inhalte) der Kante.

6.3 Umgebung

`chp-version` Funktion

`(chp-version )`

Die Funktion gibt Information über die aktuelle Version des Parsers, die geladenen Module und für jedes Modul das Datum der letzten Änderung.

`*nlpkg*` Variable

Das Lexikon liefert Kategoriesymbole zurück, mit denen die Grammatik arbeitet. Diese Variable enthält das Lisp-Package für diese Symbole. Der Parser stellt für Lexikonanfragen das Defaultpackage `*package*` zeitweise auf `*nlpkg*` um. Damit die Grammatik die Symbole des Lexikons erkennt, muß `*nlpkg*` beim Laden einer Grammatik und bei ihrer Verwendung den gleichen Wert haben.

Default ist `"USER"`

`do-with-timeout` Makro

`(do-with-timeout (time . timeoutforms) &body body)`

Führt `body` aus. Wenn `body` noch nicht zu Ende gelaufen ist stoppt es nach `time` Sekunden und gibt den Wert von `timeoutforms` zurück. Dieses Makro verwendet bei Allegro CL und auf dem TI-Explorer systemspezifische Funktionen.

Es gibt eine portable Common-Lisp Variante, die bei jedem Aufruf der Funktion `add-inactive-edge` durch Aufruf der Funktion `timer` die Zeit überprüft.

Wenn `add-inactive-edge` nicht aufgerufen wird, bricht die Common-Lisp-Variante bei Überschreiten des Timeouts allerdings nicht ab.

`with-time` Makro
`(with-time &body body)`
Das Makro `with-time` bestimmt die für die Ausführung von `body` benötigte CPU-Zeit und gibt als Werte die Zeit in Sekunden und den Wert von `body` zurück.

`k-with-time` Makro
`(k-with-time k &body body)`
Führt `body` `k`-mal aus und gibt die Zeit zurück. Damit kann man auch sehr kleine Laufzeiten, die unter der Granularität des Timers liegen, vergleichen.

6.4 Scanner

Der Zeilenscanner basiert auf dem in Abschnitt 6.5 beschriebenen ATN-Interpreter. Er enthält ATNs für eine Eingabezeile, Kategoriedefinitionen und Abkürzungen.

`scan` Funktion
`(scan string)`
Diese Funktion zerlegt `string` mit dem ATN-Scanner in Eingabeelemente.

```
> (scan "Der Berg ruft.")
("Der" "Berg" "ruft" #\.)
```

`scan-line` Funktion
`(scan-line &key mult conv)`
Diese Funktion liest eine Zeile vom Terminal und zerlegt sie mit dem ATN-Scanner in Eingabeelemente. Ist `mult` wahr, liest der Scanner mehrere Zeilen. Wenn `conv` wahr ist, werden Umlaute mit den Angaben aus `*conv-table*` umgewandelt.

`*conv-table*` Variable
Auf verschiedenen Plattformen gibt es unterschiedliche Arten, Umlaute darzustellen. Für die Portabilität von Daten kann eine beliebige ASCII-Notation für Umlaute definiert werden, z.B. /ae für ä. Die Umwandlung wird vor dem ersten Verarbeitungsschritt vorgenommen. Damit kann man Umlaute auch dann eingeben, wenn es auf einem System nicht möglich ist, Umlaute in der internen Darstellung des Lexikons einzugeben. Der Defaultwert der Umwandlungstabelle ist:

```
'(("/ue" . "\374")
  ("/oe" . "\366")
  ("/ae" . "\344")
  ("/ss" . "\337")
```

```
("/Ue" . "\334")
("/Oe" . "\326")
("/Ae" . "\304")))
```

6.5 ATN-Interpreter

Im Compilerbau definiert man Scanner häufig durch endliche Automaten [Aho et al. 86]. Für die Verarbeitung natürlicher Sprache genügt deren Mächtigkeit in einigen Fällen nicht mehr. Die Darstellung eines endlichen Automaten als FSTN (Übergangsnetz, Finite State Transition Network) erweitert man in einem ersten Schritt zum RTN (Recursive Transition Network), bei dem rekursiv weitere Netze gerufen werden. RTNs sind äquivalent zum Kellerautomaten, erkennen also kontextfreie Sprachen. Erlaubt man zusätzlich noch Zuweisungen an Register und Bedingungen für Übergänge, dann erhält man das ATN (Augmented Transition Network). ATNs haben die Berechnungsmächtigkeit der Turing-Maschine. Der Einsatz von ATNs ist sinnvoll, wenn die zu erkennende Sprache nur in wenigen Punkten von einer regulären Sprache abweicht, denn dann bleiben die Netze übersichtlich. Der von ChaPLin zum Scannen von Eingabestrings eingesetzte ATN-Interpreter ist eine Erweiterung des in [Charniak et al. 87] beschriebenen RTN-Interpreters.

6.5.1 Netzdefinition

`defnet` Makro

`(defnet name ([register]*) [description]+)`

Dieses Makro definiert ein ATN.

`name` Ein Symbol, das den Namen des neuen Netzes angibt.

Jedes `register` ist ein Symbol, das ein Register definiert. Diese Register können im Netz verwendet werden. Die in `defnet` eingeführten Variablen sind innerhalb eines Netzes lokal. Zusätzlich ist die globale Variable `-hold-` und das Register `-current-` definiert.

Die Netzbeschreibung `description` gibt an, wie die Wörter der vom ATN erkannten Sprache aussehen. Die an reguläre Ausdrücke angelehnte Notation erlaubt folgende Konstrukte:

- `(SEQ <description> <description> ...)`

  Bei einer Sequenz von Kanten bzw. Beschreibungen müssen die Konstituenten nacheinander im Wort vorkommen.

- `(OPTIONAL <description> <description> ...)`

  Die Konstituentenfolge kann im Wort vorkommen oder nicht. Der Backtrackingalgorithmus testet das Wort zuerst ohne die optionale Folge.

- `OPTIONAL* <description> <description> ...)`

  Wie `OPTIONAL`, nur daß die Konstituentenfolge beliebig oft stehen kann.

- `(EITHER <description> <description> ...)`

  Nur eine der alternativen Beschreibungen muß beim Ablauf erfolgreich durchlaufen werden.

- `(CAT <category> [:test <expr>] [:do <expr>])`

  Das nächste Eingabeelements muß vom Typ `<category>` sein und der Test darf nicht `nil` ergeben. In diesem Fall wird die Eingabe gelesen und die `:DO`-Anweisung ausgeführt. Die vordefinierte Kategorie `ANY` liest ein beliebiges Eingabeelement

- `(WORD <word-or-list> [:test <expr>] [:do <expr>])`

  Entspricht `CAT`, nur wird anstelle einer Kategorie ein bestimmtes Eingabeelement oder eine Liste möglicher Eingabeelemente angegeben.

  Dabei ist zu unterscheiden, ob die Eingabe aus einer Liste oder von einem String gelesen wird

  Liste

`table`	Nächstes Wort muß `table` sein.
`(table chair)`	Nächstes muß `table` oder `chair` sein.
`"table"`	String `"table"` muß folgen.

  String

`\#t`	Nächstes Zeichen muß `\#t` sein.
`( \#t \#a )`	Nächstes Zeichen muß `\#t` oder `\#a` sein
`"table "`	*Eines* der Zeichen `\#t` `\#a` `\#b` `\#l` oder `\#e` muß folgen.

- `(JUMP [:test <expr>] [:do <expr>])`

  Falls der Test erfüllt ist, wird die Aktion ausgeführt. Dabei wird kein Eingabezeichen gelesen.

- `(PUSH <name> [:test <expr>] [:do <expr>])`

  Falls der Test erfüllt ist, geht der Interpreter in das Netz mit Namen `<name>` über. Die `:DO` Aktionen werden erst *nach* dem Rücksprung aus dem aufgerufenen Netz ausgeführt

- `(POP <expr> [:test <expr>] )`

  Falls der Test erfüllt ist, wird das Netz verlassen. Der Rückgabewert des Netzes ist der Wert des ersten `<expr>`.

Beschreibungen, die einen `:TEST` Parameter enthalten, werden ausgeführt, wenn die Auswertung des Tests nicht `nil` ergibt. Andernfalls wird ein Backtracking ausgelöst. Wird ein optionaler `:TEST`-Parameter nicht angegeben, gilt die Bedingung als erfüllt.

Als Netzvariable sind außer den in `defnet` eingeführten lokalen Variablen die globalen Variablen `-hold-` und `-current-` definiert. Die Lisp-Ausdrücke `<expr>` verwenden diese Netzvariablen. In den Testausdrücken ist `-current-` an das

nächste zu lesende Eingabeelement gebunden (lookahead). In den `:DO`-Ausdrücken enthält `-current-` dagegen das *gerade gelesene* Eingabeelement oder im Falle von `PUSH` den vom aufgerufenen Netz zurückgegebenen Wert.

`eoinp` Funktion
`(eoinp element)`
Testet, ob das Element dem Zeichen für „Ende der Eingabe“ entspricht. Die Funktion benutzt man in der Form `(eoinp -current-)` zur Formulierung von Testbedingungen.

6.5.2 Kategorien

Die Terminale des ATNs sind die Kategorien. Die vordefinierte Kategorie `ANY` steht für ein beliebiges Eingabeelement. Sie darf nicht mit `defabbrev`, `getabbrev` oder `categoryp` überdefiniert werden.

`defabbrev` Makro
`(defabbrev category elements)`
Definiert eine Kategorie durch die Liste `elements` ihrer Elemente.
`getabbrev` Makro
`(getabbrev category)`
Gibt die Liste der Elemente, die zu der angegebenen Kategorie gehören, zurück.

`categoryp` Funktion
`(categoryp element category)`
Testet, ob das Element von der angegebenen Kategorie ist

6.5.3 Aufruf des ATN-Interpreters

`atn` Funktion
`(atn sentence &key start exhaustive trace level)`
Beim Aufruf des ATN-Interpreters wird als Argument `sentence` eine Eingabesequenz, d.h. ein String oder eine Liste übergeben. Die Keywordparameter haben folgende Bedeutung:

`start` Startkategorie, der Name des top-level-Netzes (Default: `S`)

`exhaustive` Flag, das angibt, ob der Interpreterlauf nur dann erfolgreich beendet wird, wenn die Eingabesequenz am Ende leer ist. (Default: `t`)

`trace` Trace-Level folgender Form:

- `NIL` kein Trace (Default)
- **1** nur wichtigste Informationen ausgeben,
- **2** nur wichtige Informationen ausgeben,
- **3** alle Informationen ausgeben

`level` In manchen Fällen gibt es mehrere Lösungen. Der Parameter `level` ist ein Integer, der die Nummer der Lösung nach der durch das Netz festgelegten Abarbeitungsreihenfolge angibt. Bei 1 wird die erste Lösung, bei 2 die zweite ausgegeben .... Default ist 1.

Wenn das Netz erfolgreich durchlaufen wurde, gibt die Funktion `atn` zwei Werte zurück: das Ergebnis der `POP`-Anweisung des top-level-Netzes und den Rest der Eingabesequenz (`NIL` falls `exhaustive=t`).

6.5.4 Beispiele für ATNs

Die folgenden Beispiele zeigen, wie einfache Netze für den ATN-Interpreter aussehen und wie der Interpreter aufgerufen wird. Das folgende Netz ist ein rekursives Übergangsnetz (RTN), weil es auf Registerzuweisungen verzichtet. Es erkennt Folgen vom Typ $a^n b^n$.

```
(defnet demo1 ()
  (optional
    (word a) (push demo1) (word b)))
```

Der ATN-Interpreter wird folgendermaßen aufgerufen:

```
> (atn '(a a a b b b) :start 'demo1)
T
NIL
> (atn '(a a a b b b b) :start 'demo1)
NIL
NIL
> (atn '(a a a b b b b) :start 'demo1 :exhaustive nil)
T
(a a a b b b b)
```

Wenn `:exhaustive nil` ist, dann kann das ATN auch erfolgreich durchlaufen werden, ohne ein Eingabezeichen zu lesen. Wenn aus einem String anstatt aus einer Liste gelesen werden soll, sieht das Netz folgendermaßen aus:

```
(defnet demo1-1 ()
  (optional
    (word "a") (push demo1-1) (word "b")))
> (atn "aaabbb" :start 'demo1-1)
T
""
```

Das nächste Beispiel `demo2` beschreibt die gleiche Sprache wie `demo1`. Die Analyse wird aber in einer anderen Reihenfolge durchgeführt. Man erkennt den Unterschied, wenn `:exhaustive` den Wert `nil` hat.

```
(defnet demo2 ()
  (either
    (seq (word a) (push demo2) (word b))
    (jump)))

> (atn '(a a a b b b b) :start 'demo2 :exhaustive nil)
T
(b)
```

Zur Beschreibung der Sprache $a^n b^n c^n$ reicht ein RTN nicht aus. Das Netz `demo3` erkennt Folgen vom Typ $a^n b^n c^n$. Das Subnetz `demo3-1` entspricht `demo1`, legt aber zusätzlich jedes gelesene `a` nach `-hold-`. Der Rest des Netzes `demo3` liest dann c^n. Die auskommentierte Zeile des Netzes zeigt, wie man zur Fehlersuche den Wert eines Registers ausdrucken kann.

```
(defnet demo3 ()
  (push demo3-1)
  ;;(jump :do (print -hold-))
  (optional*
    (word c :test -hold- :do (pop -hold-)))
  (pop t :test (null -hold-)))
```

Das Subnetz für $a^n b^n$:

```
(defnet demo3-1 ()
  (optional
    (word a :do (push -current- -hold-))
    (push demo3-1)
    (word b)))
```

In den `:do`-Anweisungen sind `push` und `pop` nicht Netzschlüsselwörter sondern bezeichnen die entsprechenden Lispfunktionen.

```
> (atn '(a a b b c c) :start 'demo3)
T
NIL
> (atn '(a a b b c c c) :start 'demo3)
NIL
NIL
```

Kapitel 7

Implementierung

7.1 Verlauf der Implementierung

ChaPLin ist Ergebnis einer Forschungsarbeit und ist als Hilfmittel für Forschungsarbeiten konzipiert. Im Laufe der Zeit wurden verschiedene Möglichkeiten und Varianten getestet und implementiert.

Ausgangpunkt der Entwicklung war die Beschreibung der Interlisp-Implementierung eines einfachen Chart-Parsers in [ThompsonRitchie 84]. Als erste Version reimplementierte Gerrit Burkert diesen Parser mit an Common Lisp angepaßten, effizienteren Datenstrukturen für die Chart (4/88). Zur Version 2 wurde der Parser erweitert. Unter anderem wurde die Möglichkeit zum Bottom-Up-Parsen eingeführt, die Grammatik anders repräsentiert und die Effizienz verbessert (11/89). In der dritten Version des Parsers wurde die Möglichkeit zur Verarbeitung unterschiedlicher Grammatikformalismen eingeführt (4/92).

Die jetzt vorliegende Version 3.2 von ChaPLin entstand nach einer Überarbeitung durch Mathis Löthe bis zum Januar 1995. Die einzige Erweiterung ist die Kantenverschmelzung, ansonsten wurden die internen Schnittstellen des Parsers überarbeitet, die Trennung zwischen Grammatiktyp und Parserkern verbessert, der Code innerer Schleifen optimiert und das Programm dokumentiert.

Die Überarbeitung begann am Kern des Parsers, nämlich an der Chart und der Schnittstelle zwischen den Grammatiktypen und der syntaktischen Phase. Darauf folgten der Ausgabegenerator, die Grammatiktypen selbst und die Utilities. Bei der Überarbeitung wurden die Anforderungen für die einzelnen Teile festgelegt.

7.2 Stand der Implementierung

Die zentralen Komponenten des Parsers sind bereits klar definiert und in einem stabilen Zustand. Die Anforderungen an Ausgabegenerator, Grammatik- und Lexikonschnittstelle kann man dagegen nicht auf einfache Art vollständig festschreiben. Daher sind die Begriffe und die Arbeitsweise für diese Teile noch nicht so übersichtlich spezifiziert wie für den Kern des Parsers. Die folgen-

de Aufstellung beschreibt den Zustand der einzelnen Teile und nennt mögliche Verbesserungen:

Chart Die Chart wurde von Grund auf überarbeitet und enthält keine grammatiktypabhängigen Anteile mehr. ChaPLin erzeugt für jeden Satz eine neue Chart, damit die garbage collection nach dem Parsen eines Satzes den von der Chart verbrauchten Speicherplatz wieder vollständig freigibt. Unter Allegro CL auf einer SUN-SPARC Station erwies sich dieses Vorgehen als vorteilhaft.

syntaktische Phase Die syntaktische Phase ist gründlich überarbeitet und optimiert worden. Die Trennung zwischen Kern und Grammatiktyp ist vollzogen. Nach einer Verbesserung der Lexikonschnittstelle kann man noch die Mechanismen zur Behandlung von Stringkategorien vereinfachen.

Ausgabegenerator Der Ausgabegenerator wurde neu strukturiert und die grammatiktypspezifischen Anteile ausgegliedert. Einige Vereinfachungen und Vereinheitlichungen wären aber wünschenswert:

- Erzeugung von `:alt`-Bäumen klarer definieren.
- Spezifikationsübersetzung und Steuerung von Filter und Postprozessor vereinheitlichen.
- Vereinfachung der Spezifikationsprache für die Baumerzeugung.

Lexikon Im Bereich der Lexikonschnittstelle sind noch einige Dinge verbesserungsbedürftig:

- Die Lexikonstruktur enthält aus Kompatibilitätsgründen unbenutzte überzählige Slots. Die alten Lexika sollten untersucht werden und Referenzen auf diese unbenutzten Slots entfernt werden.
- Für jedes Attribut des Lexikons existiert eine Zugriffsfunktion, die angegeben werden muß. Damit ist die Menge der Lexikonattribute von der Lexikonschnittstelle festgelegt. Ein besseres Übergabeprotokoll wäre: Jedes Lexikon erzeugt gleich die Liste `(:lex ...)`. So können Lexika verschiedene Angaben enthalten, und man kann den Attributsatz den Bedürfnissen des Grammatiktyps anpassen.
- Die Lexikonschnittstelle enthält noch `:sf`-grammatiktypspezifische Dinge (Grund s.o.).
- Die Lexikonschnittstelle sollte eine Komponente erhalten, die ein automatisches Memoizing der häufigsten Wörter betreibt. Ein Vorbild befindet sich in der Datei `textprocessor.lisp`.
- Die Lexikonschnittstelle sollte Datentypfehler in Lexikondaten abfangen, z.B. indem sie Defaultwerte liefert. Bisher geschieht dies ansatzweise im Parser selber, ansonsten führt fehlerhafte Lexikoninformation zu einem Fehlverhalten des Parsers.

Grammatiktypen Probleme bei den Grammatiktypen entstehen meist durch die Arbeitsteilung zwischen den Grammatiktypmodulen und den anderen Teilen des Parsers.

Der Grammatiktyp `:fu` – ein Grammatiktyp für Featureunifikationsgrammatiken von Petra Schmidt [Schmidt 92] – ist noch nicht an die Version 3.2 angepaßt.

Grammatik Die Mechanismen zur Definition von Grammatiken sollten in einigen Punkten noch vereinfacht werden:

- Das Verfahren zum Lesen der Regeln und Überprüfen ihrer Syntax ist noch relativ kompliziert (mehrere Überprüfungsmechanismen).
- Bisher ist es möglich, zu einer gegebenen Regelmenge den Grammatiktyp zu bestimmen. Diese Funktionalität ist meist überflüssig. Ein Verzicht darauf vereinfacht den Lademechanismus.

Beim Laden von Grammatiken werden Symbole erzeugt. Der Parser trägt sie in das Package `*nlpkg*` ein. Zu diesem Zeitpunkt werden auch Stringkategorien zu Symbolen umgeformt. Bei Lexikonaufrufen muß dieses Package auch verwendet werden, dafür muß zur Zeit die Variable `*nlpkg*` richtig gesetzt sein. Ein flexiblerer Mechanismus wäre wünschenswert.

Inkrementelles Parsen Der inkrementelle Parser verwendet systemspezifische Funktionen und Konstantendefinitionen zur Ansteuerung des Terminals. Diese müssen an das jeweilige System angepaßt werden.

Die augenblickliche Version des inkrementellen Parsers parst nach jedem gelesenen Lexem und reagiert erst dann wieder auf Eingaben, wenn dieser Teil des Parsevorgangs abgeschlossen ist. Bei Allegro CL 4.2 auf einer SUN-SPARCstation 10 kann man problemlos damit arbeiten. Vermutlich ist der Parser schnell genug oder die Eingabe wird sinnvoll gepuffert.

Sollte es auf einer Plattform nötig sein, auch während des Parsens die Tastatur abzufragen, dann muß der Aufruf des inkrementellen Scanners

```
#+<Plattform>(when *incremental* (read-input-char))
```

an einer geeigneten Stelle im Parser eingefügt werden, z.B. in die Funktion `add-inactive-edge` . Die Funktion `parse-incremental` muß dann die globale Variable `*incremental*` entsprechend belegen.

Utilities Die Funktionen zur graphischen Ausgabe von Parsebäumen sind plattformabhängig. Es existieren ältere Versionen für TI-Explorer und CLX. Bei Bedarf sollten Ausgaben für weitere Plattformen hinzugefügt werden, z.B. für CLIM oder Garnet.

7.3 Weiterentwicklung und Ausblick

Ziel der Weiterentwicklung muß es sein, ein Werkzeug für die Verarbeitung natürlicher Sprache zu erhalten. Da verschiedene Anwendungen unterschiedliche Anforderungen an ihren Parser stellen, ist die Flexibilität des Parsers besonders wichtig.
In einigen Teilbereichen ist dieses Ziel bereits erreicht:

- Verschiedene Grammatiken und Grammatiktypen können einfach erstellt und ausgetauscht werden.
- Der Parser behandelt unterschiedlich vorverarbeitete Eingaben.
- Die Ausgabeform läßt sich flexibel den Bedürfnissen der Anwendung anpassen.
- Die genannten Möglichkeiten sind weitgehend frei miteinander kombinierbar.

Dennoch bleibt auch für die Zukunft noch einiges zu tun. Außer einer Korrektur der im vorigen Abschnitt genannten Probleme der Einzelkomponenten sind noch folgende generelle Verbesserungen wünschenswert:

- Einige Spezifikations- und Steuersprachen sind noch zu kompliziert und zu mühsam zu erlernen, vor allem im Bereich der Ausgabegenerierung.
- Der Attributierungsmechanismus ermöglicht es, Teile der semantischen Analyse schon in der Ausgabephase des Parsers durchzuführen. Dabei steht die kompakte Darstellung des Parsewalds in der Chart zur Verfügung.

  Eine erweiterte Attributierungsspezifikation nach Vorbildern aus dem Compilerbau (vgl. syntaxgesteuerte Übersetzung in [Aho et al. 86]) würde diese Möglichkeiten für den Anwender deutlicher machen.
- Der Leistungsumfang von ChaPLin sollte erweitert werden, d.h. Aufgaben, die jetzt noch von anderen Programmen übernommen werden, wie das Parsen größer Textkorpora oder die Verwendung verschiedener Lexika sollten von standardisierten Zusatzbausteinen des Parsers übernommen werden.
- Bei den Hilfsprogrammen kann eine graphische Anzeige von Ergebnissen die Untersuchung des Parsevorgangs vereinfachen und damit den Nutzen von ChaPLin für den Einsatz in der Spachverarbeitung erhöhen.

Trotz aller Wünsche und Verbesserungsvorschläge ist ChaPLin bereits in der augenblicklichen Version 3.2 eine stabile und flexible Komponente für den Aufbau eines sprachverarbeitenden Systems. Daher:

Viel Spaß und Erfolg mit ChaPLin 3.2 !!!

Anhang A

Beispiele

A.1 Beispiellexikon

Dieses Testlexikon enthält die Einträge für das Beispiel in Kapitel 3. Es entnimmt die Einträge aus einer Liste von Datensätzen, die bereits in dem vom `:sf`-Grammatiktyp geforderten Format abgelegt sind. Für den Einsatz in einem Anwendungsprogramm kann der Parser an externe Lexika und ggf. eine Morphologiekomponente angeschlossen werden.

```
(in-package "USER")

(if (fboundp 'chp:defmod)    ;;;;   VERSION STRING
    (chp:defmod :lexicon "Lexikonschnittstelle Beispiel" "13-DEZ-95")
    (error "** file PARSER not yet loaded **"))

;;; sample data
(setq *sample-lex-data*
  '(("Der" ("Der" det "DER" 1 nil "*DER"))
    ("Berg" ("Berg" n "BERG" 1 ((num sg) (gen mask)) "*BERG"))
    ("ruft" ("ruft" vf "RUFT" 1 ((time praes) (pers p3) (num sg)) "*RUFT"))
    (#\. ("." punkt "." 1 nil nil))))

;;; define lexicon format
(setq *sample-lex*
  (chp:make-lex
   :entries #'(lambda (word &optional igncap)
                (cdr (assoc word *sample-lex-data* :test #'equal)))
   :form    #'first
   :cat     #'second
   :word    #'third
   :conf    #'fourth
   :attr    #'fifth
   :sem     #'sixth))

(setq chp:*lexicon* *sample-lex*)
```

A.2 Beispielgrammatik

Diese Grammatik ist ebenfalls für die Beispiele aus Kapitel 3 und einfache Tests vorgesehen. Die Regeln entsprechen dem in Abschnitt 5.5 beschriebenen Format des `:sf`-Formalismus. Sie enthalten aber keine Anweisungen zur Featurebehandlung, so daß es sich hier um eine rein kontextfreie Grammatik handelt.

```
(in-package "USER")

;;;;   VERSION STRING
(if (fboundp 'chp:defmod)
    (chp:defmod :grammar "Deutsche Beispielgrammatik :sf" "13-DEZ-95")
    (error "** file PARSER not yet loaded **"))

;;; Grammar definition

(setq *sample-grammar*
      (chp:define-grammar
        '((S -> (NP VP PUNKT) 1 NIL NIL NIL -)
          (NP -> (DET N) 1 NIL NIL NIL -)
          (NP -> (DET ADJ N) 1 NIL NIL NIL -)
          (VP -> (VF)  1 NIL NIL NIL -)
          (VP -> (VF NP)  1 NIL NIL NIL -))
        :type :sf
        :ignore ()))

(setq chp:*grammar* *sample-grammar*)
```

Anhang B

Verzeichnis der zugehörigen Dateien

Dieser Anhang beschreibt die Dateien des Parsers, deren Funktionalität in diesem Bericht dokumentiert oder erwähnt ist. Dateien, die darüber hinaus gehende Erweiterungen darstellen, werden hier nicht aufgeführt. Sie sollten jedoch in die Datei `doc/files<version>.txt` eingetragen werden.

B.1 Unterverzeichnisse

Der Quellcode liegt im Hauptverzeichnis von ChaPLin.

Binaries `bin`
: In diesem Verzeichnis liegen die kompilierten Dateien von ChaPLin. Sie werden von `load-module` automatisch verwaltet und bei Bedarf neu kompiliert.

Dokumentation `doc`
: Die Dokumentationsdateien in diesem Verzeichnis sind ASCII-Texte, die bei neuen Versionen laufend aktualisiert werden sollten.

Beispiele `Examples`
: Dieses Verzeichnis enthält verschiedene Grammatiken und Lexikonschnittstellen

B.2 Codedateien von ChaPLin

Lader `load-parser.lisp`
: Die Startdatei enthält einen Aufruf der Funktion `load-module`, die den Parser lädt und bei Bedarf (teilweise) kompiliert. In diese Datei werden alle plattform- und installationsabhängigen Einstellungen eingetragen.

Modulverwalter `load-module.lisp`
: Implementation der Funktion `load-module`. Dieser Modulverwalter kom-

piliert die Dateien nach Bedarf und legt die Binärdateien in das Unterverzeichnis `bin`.

Parser `parser-<version>.lisp`
Der Parserkern enthält die Definitionen von Chart und Parserfunktionen und die Mechanismen zur Definition von Grammatiktypen, Grammatiken und Lexika.

Grammatiktypen `grammar-types-<version>.lisp`
Diese Datei definiert die Grammatiktypen für einfach kontextfreie (`:cf`) und flat feature (`:sf`) Grammatiken.

Utilities `utilities-<version>.lisp`
Spezielle Aufrufvarianten für den Parser, Zeitmessung, Timeout und Chartanalysen.

Scanner `scan-line<version>.lisp`
Schnittstellenfunktionen und Netzdefinitionen für den Zeilenscanner.

ATN-interpreter `atn-interpreter-<version>.lisp`
ATN (Augmented Transition Network) Interpreter

Inkrementeller Parser `incremental-<version>.lisp`
Liest Eingabe zeichenweise und parst sobald möglich. Enthält einen eigenen, inkrementellen Scanner, der Token für Token an den Parser übergibt. Der inkrementelle Parser verwendet plattformabhängige Funktionen.

B.3 Dokumentationsdateien

Die Dokumentationsdateien liegen normalerweise im Unterverzeichnis `doc`. Sie enthalten Angaben über den aktuellen Zustand der Installation und sollten daher nach Änderungen auf den neuesten Stand gebracht werden.

Übersicht `README`
Diese Datei sollte im Hauptverzeichnis liegen und enthält Information über die aktuelle Version.

Dateiliste `files-<version>.text`
Das Dateiverzeichnis enthält eine Beschreibung aller Dateien und ihrer Bedeutung.

Fehlerliste `error-codes.txt`
Der Chartparser benutzt für seine internen Fehlermeldungen eine eigene Numerierung. Die Fehlerliste für ChaPLin 3.2 ist in Anhang C abgedruckt. Die Datei enthält ein Verzeichnis aller Fehlernummern und Fehlermeldungen und für jeden Fehler einen kurzen Hinweis auf Bedeutung und typische Ursachen.

Anhang C

Fehlertabelle

ChaPLin erzeugt numerierte Fehlermeldungen durch Aufruf der Funktion `chp-error`. Die Aufstellung enthält die Fehlermeldungen aller in diesem Bericht beschriebenen Teile des Parser und Hinweise auf Ursache und Lösungsmöglichkeiten. Fehlermeldungen in eigenen Erweiterungen sollten in der Datei `doc/errors.txt` dokumentiert werden.

00 Ladevorgang

01 Ladefunktion `load-module` nicht gefunden.

02 Grammatikdatei kann nicht vor dem Parser geladen werden.

03 Lexikondatei kann nicht vor dem Parser geladen werden.

04 Grammatiktyp kann nicht vor dem Parserkern geladen werden.

10 Initialisierung

11 Wert für den Parameter `find` im Top-Down-Modus unzulässig.

20 Lexicon

Die Fehler 27-29 werden von einem Patch zur Kompatibilität mit älteren Lexika erzeugt.

21 Lexikoneinträge `<item>` sind inkonsistent.

22 Im Lexem `<item>` fehlt die Kategorie.

27 Probleme beim Zugriff auf die Kategorie.

28 Probleme beim Zugriff auf das Wort (Grundform).

29 Probleme beim Zugriff auf die ursprüngliche Wortform.

30 Chartzugriff

31 Undefinierter Kantentyp `<edge-type>` bei der Expansion des Makros `insert-edge` . Zulässig sind `:active`, `:inactive` oder der Name der Konstruktorfunktion eines vom Grammatiktyp definierten Lesartkantentyps. Tritt meist beim Laden eines Grammatiktyps auf.

32 Unzulässige Knotennummer `<number>`. Die Knotennummer ist entweder kein Integer oder liegt außerhalb des zulässigen Bereichs.

33 Unzulässige Kantennummer `<number>`. Die Kantennummer ist entweder kein Integer oder liegt außerhalb des zulässigen Bereichs.

40 Ausgabegenerator

Die Fehler 41-43 entstehen durch Datenfehler in der Chart, 45-47 durch fehlerhafte Spezifikationen.

41 `chart-subtree` `<edge>` ist keine Kante.

42 `chart-subtree` Terminalkante `<edge>` darf keine Ignore-Kante sein. Der Ausgabegenerator kann ein Terminalsymbol nicht durch seine Konstituenten ersetzen. Der Fehler kann auch für einen Struktureintrag `(:struct ...)` auftreten.

43 Unzulässiger Inhalt `<content>` einer Terminalkante. Der Inhalt muß mit `:lex` oder `:struct` beginnen.

45 Vom Grammatiktyp `<type>` erzeugte Ausgabespezifikation `<spec>` ist undefiniert.

46 Fehler in der Baumspezifikation.

49 `:alt`-Baum kann nicht erzeugt werden.

50 Grammatik

51 Syntaxfehler in der Regel `<rule>`.
Regel entspricht nicht der Regelsyntax des Grammatiktyps.

52 Grammatiktyp `<type-name>` unbekannt.

53 Kein passender Grammatiktyp verfügbar.
Der Parser sucht nach einem Grammatiktyp zur Regelmenge. Der beabsichtigte Typ ist nicht definiert oder die Grammatik ist fehlerhaft.

54 Unzulässiger Wert für den Parameter `mode`. Erlaubt sind `:td` für top-down und `:bu` für bottom-up.

70 Grammatiktyp `:cf`

71 Kurzspezifikation `<type>` im `:cf`-Grammatiktyp nicht definiert.

80 Grammatiktyp `:sf`

Die Fehler 85-88 entstehen beim Aufbau der Semantikstruktur.

81 Kurzspezifikation `<type>` im `:sf`-Grammatiktyp nicht definiert.

82 Syntaxfehler in Regel `<rule>`.

85 Zugriff auf nicht vorhandene Kategorie in `(&SEM cat)`.

86 Zugriff auf nicht vorhandenes Feature in `(&ATTR fea)`.

87 Zugriff auf nicht vorhandenes Feature in `(&ATTR (cat fea))`.

88 Zugriff auf nicht vorhandene Kategorie in `(&ATTR (cat fea))`.

110 Utilities

111 Parserkern muß vor den Utilities geladen werden.

112 graphische Ausgabe in dieser Implementierung nicht möglich.

113 Die Baumform `<tree>` kann nicht angezeigt werden.

114 Unzulässige Knoten `<vertex>` für den Parameter `:from` in `display-chart` .

117 Unzulässiger Kantentyp `<type>` in Funktion `display-chart`. Erlaubt sind `:all`, `:active`, `:inactive` und `:tree`.

130 Zeilenscanner

131 Fehler im ATN-Interpreter: unbekanntes ATN Schüsselwort.

132 Fehler in der Netzbeschreibung: falsch plaziertes `POP`.

133 Netz in `PUSH` Anweisung unbekannt.

134 Keine weiteren Zeichen in der Eingabesequenz.

Anhang D

Inhaltverzeichnis des Codes

Die beiden wichtigsten Dateien von ChaPLin 3.2 sind der Parser und die Grammatiktypdefinition. In diesen beiden Dateien ist im Code eine Gliederung als Suchhilfe angebracht. Die Teile 1-5 gehören zum Parserkern, Teil 6 zu den Grammatiktypen. Zum schnelleren Überblick ist hier das Inhaltsverzeichnis dieser Gliederung abgedruckt.

```
;;;; 1. basic defs
;;;;    1.1 package declaration
;;;;    1.2 module defs
;;;;    1.3 global parameters
;;;; 2. chart
;;;;    2.1 definitions
;;;;    2.2 interface functions
;;;; 3. parser
;;;;    3.1 interface functions
;;;;    3.2 fill chart with terminals
;;;;    3.3 parsing loop
;;;; 4. output generation
;;;;    4.1 main function
;;;;    4.2 form output from edges
;;;;    4.3 create tree from edge
;;;;    4.4 get success edges
;;;;    4.5 return parsed fragments
;;;;    4.6 expand edges to tree edges
;;;;    4.7 create alt trees
;;;; 5. interfaces
;;;;    5.1 grammar types
;;;;    5.2 grammars
;;;;    5.3 lexicon
;;;; 6. grammar types
;;;;    6.1 cf
;;;;    6.2 sf
```

Glossar

Ableitung (Derivation) Ergebnis einer oder mehrerer Regelanwendungen.

Aktivierung Bei der Aktivierung der Chart werden für Regeln, deren Anwendung möglich erscheint, aktive Kanten in die Chart eingetragen.

Attribut Information, die die Syntaxanalyse nicht beeinflußt.

Chart ChaPLin repräsentiert die Zwischenschritte der Syntaxanalyse als Kanten eines gerichteten Pseudographen [Harary 74], der Chart. Die Knoten der Chart stehen für die Zwischenräume im Satz, die Kanten überspannen also einen bestimmten Teil des Satzes.

Eingabeelement Eingabeelemente sind Wortformen, Zahlen oder Satzzeichen. Der Scanner zerlegt eine Zeichenkette (String) in Eingabelemente. Im Compilerbau entsprechen die Eingabeelemente den Tokens [Aho et al. 86].

Feature Information, die neben der Kategorie zur Steuerung der Syntaxanalyse herangezogen werden kann.

Grammatik Eine Menge von Ableitungsregeln [Aho et al. 86].

Grammatikformalismus Ein Grammatikformalismus definiert Form und Bedeutung der Grammatikregeln.

Grammatiktyp Implementation eines Grammatikformalismus zur Verwendung für ChaPLin.

Graph Ein Graph besteht aus Knoten, die durch Kanten verbunden sind (siehe [Harary 74]). Sowohl die Chart als auch die verwendeten Baumstrukturen werden hier als Graphen repräsentiert.

Grundform Die Grundform (auch Nennform) ist die unflektierte Form eines Worts (z.B. Grundform “Haus” zur Wortform “Häuser”).

Grundregel Das von ChaPLin verwendete Ableitungsverfahren der Syntaxanalyse mit einer aktiven Chart.

Kategorie Die Kategorie ist die wichtigste Information bei der Syntaxanalyse (Grammatiksymbol). Bei Wörtern ist die Kategorie die Wortart, bei Nichtterminalsymbolen benennt sie das grammatikalische Konstrukt.

Kantenverschmelzung Wenn mehrere Ableitungswege zum gleichen Grammatiksymbol führen, wird nur eine Kante angelegt.

Lesart Bei mehrdeutigen Grammatiken oder bei Mehrdeutigkeiten bei der lexikalischen Analyse der Eingabesequenz kann es mehrere Ableitungswege für ein Symbol geben. Diese Ableitungswege heißen Lesarten.

Lexikon Das Lexikon bestimmt zu einer Wortform die Grundform und die morpho-syntaktischen Merkmale.

Modus Es gibt zwei Parsemodi: Beim *bottom-up*-Parsen beginnt die Konstruktion des Syntaxbaums an den Blättern, beim *top-down*-Parsen an der Wurzel [Aho et al. 86].

Natürliche Sprache Dieser Begriff wird für die Sprachen des Menschen im Gegensatz zu formalen Sprachen oder Programmiersprachen benutzt.

Parser Programm, das entscheidet, ob der Eingabesatz in die Beschreibung der Grammatik fällt und im Erfolgsfall dem Eingabesatz einen oder mehrere Syntaxbäume zuordnet.

Regel Bestandteil der Grammatik.

Scanner Der Scanner zerlegt die zunächst als Zeichenkette vorliegende Eingabe in Eingabeelemente.

Schlingen Kanten, bei denen Anfangs- und Endknoten gleich sind.

Semantik Bedeutung sprachlicher Ausdrücke [Aho et al. 86].

Spezifikation Einstellungen an ChaPLin werden mit Spezifikationen vorgenommen. Spezifikationen sind normalerweise Listen von Schlüsselwörtern und zugeordneten Werten.

Startsymbol Zielkategorie der Grammatik. Die Syntaxanalyse ist erfolgreich, wenn aus der Eingabesequenz ein (das) Startsymbol abgeleitet werden kann.

Syntax Lehre von der Anordnung von Wörter zu Sätzen. Mögliche Anordnungen werden i.A. durch ein System von Regeln (Grammatik) beschrieben.

Syntaxbaum (auch Parsebaum, Strukturbaum, Ableitungsbaum) Ergebnis der Syntaxanalyse. Der Syntaxbaum beschreibt die syntaktische Struktur eines Satzes.

Wortform (Vollform) Möglicherweise flektierte Form eines Wortes.

Index

Literaturverzeichnis

[Aho et al. 86] A. V. Aho, R. Sethi und J. D. Ullman. *Compilers : principles, techniques, and tools*. Addison-Wesley, Reading, Mass., 1986. 796 S.

[Allen 87] J. Allen. *Natural Language Understanding*. Benjamin Cummings Publishing Company Inc., 1987.

[Charniak et al. 87] E. Charniak, C. K. Riesbeck, D. V. MacDermott und J. R. Meehan. *Artificial intelligence programming*. Erlbaum Associates, Hillsdale, N. J. [u.a.], 2nd ed. edition, 1987. 533 S.

[GazdarMellish 89] G. Gazdar und C. Mellish. *Natural Language Processing in LISP:An Introduction in Syntactic Processing*. Addison Wesley, 1989.

[Harary 74] F. Harary. *Graphentheorie*. R.Oldenbourg, 1974.

[Kay 80] M. Kay. *Algorithm Schemata and Data Structures in Syntactic Processing*. XEROX Corporation CSL-80-12, 1980.

[King 83] M. E. King. *Parsing Natural Language*. Academic Press, 1983.

[Schmidt 92] P. Schmidt. Ein Chartparser für eine feature-orientierte Grammatik. Studienarbeit Nr. 985, Institut für Informatik, Universität Stuttgart, 1992.

[Seiffert 87] R. Seiffert. Erarbeitung von Parsingstrategien für Unifikationsgrammatiken mit ID-LP-Regeln. Studienarbeit Nr. 591, Institut für Informatik, Universität Stuttgart, 1987. 97 Bl.

[Seiffert 89] R. Seiffert. Chart-Parsing of Unification-Based Grammars with ID/LP-Rules. LILOG Report 22, IBM Deutschland GmbH WT LILOG / Dept 3504, P.O. Box 80 08 80, D-7000 Stuttgart 80, Germany, 1989.

[Steele 90] G. J. Steele. *Common LISP: The Language*. Digital Press, second edition, 1990.

[ThompsonRitchie 84] H. Thompson und G. Ritchie. Implementing Natural Language Parsers. In T. O'Shea und M. Eisenstadt (Hrsg.), *Artificial Intelligence*, S. 245–300. Harper & Row, 1984.

[Varile 83] G. Varile. Charts: a Data Structure for Parsing. In M. King (Hrsg.), *Parsing Natural Language*, S. 73–90. Academic Press, 1983.