Inhaltsverzeichnis

1 Einführung

1.1 Was ist Text-to-Speech Synthese?

Text-to-Speech Synthese bezeichnet die Umsetzung von beliebigem Text, dargestellt in gewöhnlicher, computerlesbarer Orthographie, in entsprechende künstliche Sprachsignale, welche als digitale Audiodaten repräsentiert werden. Im Deutschen wird dieser Vorgang oft als *Sprachsynthese aus Text* oder *Vollsynthese* bezeichnetet. In der Vorliegenden Arbeit wird der Begriff *Text-to-Speech*, welcher gebräuchlicher Weise zu TTS abgekürzt wird, aus dem angelsächsischen Sprachgebrauch übernommen.

Ein Überblick über die TTS-Synthese ist in Abbildung 1 zu finden.

1.2 Wozu Text-to-Speech Synthese?

Der Wunsch des Menschen nach einer "sprechenden Maschine" ist nicht neu: Bereits um 1790 erfand Wolfgang von Kempelen die erste Aparatur, welche Vokale und einfache Vokal-Konsonant-Paare aussprechen konnte. In greifbare nähe gerückt ist die Erfüllung dieses Wunsches, aber erst durch die Erfindung des Computers. Die Möglichkeit der natürlichsprachlichen Mensch-Maschine-Kommunikation, wie sie auch oft in Science-Fiction Romanen beschrieben wird, wurde vor allem in den 70er-Jahren, als sehr viel Intelligenz und Kapital in die Erforschung von *künstlicher Intelligenz* gesteckt wurde, massiv

vorangetrieben. Die fehlende bzw. teure Rechenleistung verhinderte damals den Durchbruch.

Heute ist die damalige Euphorie der Ernüchterung gewichen: Für die Mensch-Maschine-Kommunikation stehen inzwische Leistungsfähige, intuitiv und vor allem schnell bedienbare Benutzungsoberflächen zur Verfügung, welche die inzwischen weiter vorangeschrittenen Möglichkeiten der natürlichsprachlichen Kommunikation mit Maschinen in Nischen drängt. Natürlichsprachliche Mensch-Maschine-Schnittstellen werden in erster Linie für die Bereiche entwickelt, in denen der visuelle Weg versperrt bleibt:

- Für Menschen mit Sehbehinderungen, die sich beispielsweise von einem TTS-System ein Buch vorlesen lassen können.
- Elektronische Telefonauskünfte, wie z.B. Fahrplanauskunft.
- Bereichen in denen die BenutzerIn visuell oder motorisch bereits voll beschäftigt ist, wie beispielsweise bei der Steuerung eines Automobils.
- Digitale Assistenten, welche für eine konfortable visuelle Schnitstelle schlicht zu klein sind.

Prinzipiell stellt sich die Frage, ob es in einer Zeit wachsender sozialer, ökologischer und ökonomischer Probleme nicht wichtigere Aufgaben zu lösen gibt, als Maschinen das Sprechen beizubringen. Grundlegende Zusammenhänge zwischen Informationstechnologie und der Zukunftsfähigkeit einer Gesellschaft finden sich u.a. in [20].

1.3 Aufbau eines TTS-Systems

Der in diesem Abschnitt beschriebene Aufbau eines TTS-Systems orientiert sich an [9] bzw. [10] und wurde bereits von Diemo Schwarz im Rahmen des Hauptseminars "Sprachsynthese" im WS 1996/97 am IMS vorgestellt (vgl. [11]). Aus linguistischer Sicht läßt sich ein TTS-System, wie in Abbildung 1 dargestellt, im wesentlichen in drei Teile zerlegen: Textvorverarbeitung, linguitische Verarbeitung und Synthese.

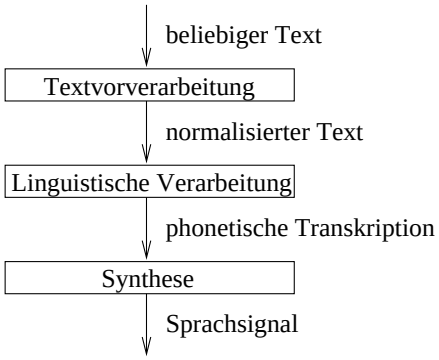

Abbildung 1: Die Grobstruktur der Text-to-Speech Synthese

1.3.1 Die Textvorverarbeitung

Die *Textvorverarbeitung* ist zeitlich an erster Stelle der TTS-Synthese angesiedelt. Sie hat die Aufgabe, den zur Synthese anstehenden Text zu *normalisieren*. Uneingeschränkter Text enthält außer regulären Worten auch

- Formatierungen und Gliederungen
- Worte in besonderer Schreibweise
- Zahlen

- Abkürzungen
- Sonderzeichen
- Interpunktion

Um die nachfolgenden Analyse- und Syntheseschritte von der Behandlung dieser Besonderheiten zu entlasten, muß die Textvorverarbeitung den rohen Text in gefilterter Form weitergeben. Sie ist u.a. dafür zuständig Zahlen, Abkürzungen und Sonderzeichen zu expandieren, sowie Interpunktionszeichen zu separieren.[1] Der menschliche Leser setzt für einige dieser Augfgaben in besonderem Maße Welt- und Kontextwissen ein. Maschinelle Textvorverarbeitunssysteme sind in der Regel mit diversen Heuristiken bzw. statistischen Methoden ausgerüstet, um die o.g. Probleme zu Bearbeiten.

1.3.2 linguistische Verarbeitung

Der allgemeine Fall der *linguistische Verarbeitung* ist in Abbildung 2 dargestellt.

Die *morphologische Analyse* ist bei flexions-, derivations- und Kompositionsreichen Sprachen wie dem Deutschen besonders wichtig. Sowohl bei einer regel- als auch bei einer lexikonbasierten *Graphem-Phonem Konversion* ist das Wissen über die innere Wortstruktur unerlässlich. Bei einem regelbasierten Ansatz deshalb, weil sich (deutsche) Ausspracheregeln an ihr orientieren. Bei Verwendung eines Aussprachelexikons, weil die Speicherung sämtlicher Kompositions- Derivations- und Flexionsformen nicht möglich ist.[2]

[1] Diese Spezifikation der Augaben der Textvorverarbeitung ergibt sich nicht zwangsläufug (siehe 3 und [15] Kapitel 4

[2] vgl. hierzu [5] Kapitel 3 *Die Wortstrukturanalyse als Grundlage der Textumsetzung*

Die *syntaktische Analyse* ermittelt die syntaktische Struktur einer Äußerung und liefert damit wichtige Informationen zur Disambiguierung der morphologischen Analyse, zur Prosodiebestimmung und liefert die Grundlage für eventuelle *höhere Analysen* (Semantk und Pragmatik).

Ausgehend vom Satzmodus (Aussage-, Fragesatz) und von der Betonung, die vom semantischen und pragmatischen Kontext abhängt, wird die *Intonation*, also der Satzakzent bestimmt. Die *Phrasierung*, also die Bestimmung der Satzpausen unterschiedlicher Länge wir in erster Linie von der Syntax bestimmt.

Aus den Phrasierungs- und Intonationsinformationen erzeugt die *Prosodiesteuerung* die konkreten suprasegmentalen Merkmale der Äußerung. Insbesondere sind das die Sprachgrundfrequenz und die Lautdauer der einzelnen Segmente.

1.3.3 Synthese

Die Synthesestufe erzeugt aus der abstrakten phonetischen Transkription das digitale Sprachsignal (Waveform), welches dann, mittels eines Digital/Analog-Wandlers, auf einem Lautsprecher ausgegeben werden kann. Um Natürlichkeit zu erreichen, müssen Koartikulationseffekte zwischen den Sprachsegmenten realisiert werden.

Für die Synthese gib es zwei unterschiedliche Ansätze: Bei der *parametrischen Synthese* kann entweder akustisch, d.h. mit einem Formantensynthesizer, oder artikulatorisch, durch simulation der menschlichen Sprechwerkzeuge (Vokaltrakt, Nasaltrakt u.s.w.) vorgegangen werden. Dazu wird die Phonetische Transkription zunächst in eine Folge von Parametersätzen (Vektoren) für

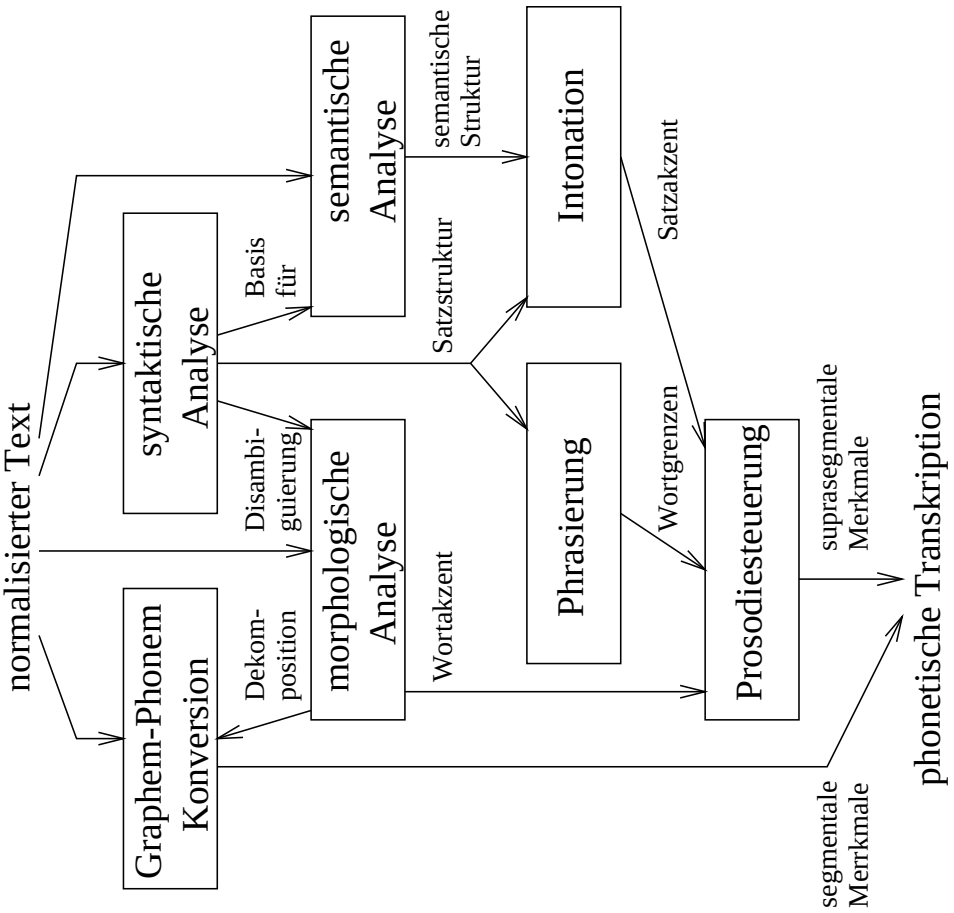

Abbildung 2: linguistische Verarbeitung

einen Sprachsynthesizer umgesetzt, welcher daraus das Sprachsignal erzeugt. Koartikulationseffekte müssen hierfür explitit generiert und in die Parametersätze aufgenommen werden.

Die zweite Möglichkeit stellt die *konkatenative Synthese* dar. Sie arbeitet mit einer Datenbank, in der Sprachaufnahmen einer Sprecherin oder eines Sprechers gespeicher sind, und extrahiert daraus die zur Generierung der Äußerung notwendigen Einheiten. Diese Einheiten sind in heutigen Synthesesystemen meist Diphone. Es können aber prinzipiell Sprachsegmente

beliebiger Länge verwendet werden. Der Aufwand zur Selektion des jeweils passenden natürlichsprachlichen Ausschnitts wird dementsprechend größer. Bei konkatenativen Verfahren müssen gegebenenfalls die suprasegmentalen Eigenschaften wie Dauer und Sprachgrundfrequenz durch Nachbearbeitung des Sprachsignals (z.B. PSOLA-Resynthese) realisiert werden.

Beim sogenannten *unit selction* Verfahren (vgl. [23] [16] zitiert in [1]) wird versucht, Segmente eines menschlichen Sprechers auszuwählen, welche den benötigten suprasegmentalen Parametern bereits entsprechen. Eine Nachbearbeitung durch Resynthese kann dann unter Umständen ganz entfallen.

2 Das Festival Speech Synthese System

In diesem Abschnitt soll ein kurzer Überblick über das *Festival Speech Synthese System* gegeben werden. Eine ausführlich Beschreibung findet sich in [1]. Die neuesete Version von Festival sowie aktuelle Informationen über das Projekt sind auf der Festival Homepage [21] zu bekommen.

Das von Alan W Black, Paul Taylor und Richard Carley am *Centre for Speech Technology Research* an der *University of Edinburgh* entwickelte *Festival Speech Synthese System* bietet einerseits ein allgemeines Gerüst zum Aufbau von TTS-Systemen und ist auf der anderen Seite bereits mit vielen beispielhaften Modulen ausgerüstet, die es zu einem kompletten TTS-System machen.

2.1 Die Benutzung von Festival

Festival bietet sowohl der BenutzerIn als auch dem Progrmmierer eine Fülle von Schnittstellen (Scheme, Emacs, Client-server...). Um sich mit dem System vertraut zu machen und wärend der Implemetierungsphase wird man wohl meist auf den Kommandointerpreter zurückgreifen. Die ersten Schritte zu dessen Benutzung und seine grundlegende Funktionalität wird deshalb im folgenden kurz beschrieben.

Wird Festival ohne Argumente gestartet meldet er sich mit einer Versions- und Copyrightinformation und dem Eingabeaufforderung `'festival>'` des Interpreters. emph'$' bezeichnet die Eingabeaufforderung (Prompt) des Betriebssystems.

```
$ festival
Festival Speech Synthesis System 1.2.1 September 1997
Copyright (C) University of Edinburgh, 1996,1997. All rights reserved.
For details type `(festival_warranty)'
festival>
```

Da Festival die Readline-Bibliothek von GNU benutzt, stehen auf dieser Ebene viele Möglichkeiten der Kommandozeilen-Editierung wie man es von *Emacs* oder *bash* kennt. Insbesondere wird eine History über die Cursortasten, sowie Funktion-, Variablen- und Dateinamenvervollständigung mittels `TAB`-Taste zur Verfügung gestellt.

Durch die Einbindung des SIOD Schemeinterpreters unterszützt Festival auf diese Ebene die meisten Standard-Schemekommandos. Ein Manual für Scheme findet sich auf [22]. Leider sind nicht alle dort beschriebenen Funktionen auch in Festival implementiert. Welche Schemefunktionen tatsächlich zur Verfügung gestellt werde läßt sich auf Unixebene durch Eingabe von

```
$ grep init_subr *.cc|more
```

im Verzeichnis `src/arch/siod-3.0/` herausfinden. Befindet sich die benötigte Funktion nicht unter den dort angegebenen, muß sie selbst implemetiert werden (in Scheme oder C++). Wenn man der Meinung ist, diese Funktion würde das Gesamtsystem wirklich bereichern, sollte man nicht davor zurückschrecken dies mit Allan Black auszudiskutieren. Für Verbesserungvorschläge, sowie noch nicht entdeckte Fehler findet man bei ihm immer eine offenes Ohr; für bereits entdeckte Verbesserungsmöglichkeiten hat er immer eine Lösung in der Tasche.

Außer den Standard Schemefunktionen stehen eine Menge Funktionen zur Sprachsynthese zur Verfügung. Welche das sind und wie man Hilfe zu den einzelnen Funktionen bekommt ist in [1] Kapitel 6.2 und 6.3 beschrieben. Mit `(exit)` kann Festival wieder verlassen werden.

2.2 Die Äußerungen in Festival

Die grundlegende Datenstruktur in Festival ist die C++ Klasse *Utterance* aus der *Edinburgh Speech Tools Library* (vgl. [2]). Sie besteht im wesenlichen aus einer Liste von benannten Datenströmen (streams) und hat einen bestimmten Typ. Jeder Strom besteht aus einer geordneten Liste von Eintragungen (items). Eintragungen haben Merkmale (features) und Werte (values) und können auch auf Eintragungen in anderen Strömen verweisen. Ein Besipiel für solch eine Utterance-Struktur erhält man indem man auf die Eingabeaufforderung von Festival `(test_show ``der 2. April'')` eingibt. Hier dir (gekürzte) Ausgabe:

```
festival> (test_show "der 2. April")
separator ;
nfields 1
# IForm
```

```
0 26  ""; a2 type Text iform "\"der 2.  April\"";
# Token
0 26  der; a411 punc 0 whitespace "" prepunctuation "";  Word 424;
0 26  2; a415 punc "" whitespace " " prepunctuation "" token_pos ordinal; Word 428;
0 26  April; a419 punc 0 whitespace "  " prepunctuation ""; Word 432;
# Word
0.713 26  der; a424 pbreak NB pos PRON; Token 411; Phrase 437; Syllable 447;
1.193 26  zweite; a428 pbreak NB pos NUM; Token 415; Phrase 437; Syllable 463 483;
1.601 26  April; a432 pbreak B pos N; Token 419; Phrase 437; Syllable 495 503;
# Phrase
1.601 26  B; a437; Word 424 428 432;
```

Die Ausgabe zeigt die Äußerung vom Typ `Text` (in der Ausgabe nicht ersichtlich). Es wurden hier aus Platzgründen nur die ersten vier Ströme abgedruckt. Jeder Gartenzaun (`#`) zeigt den Start eines neuen Stroms an. Der String hinter dem Gartenzaun ist der Name des Stroms. Jeder Eintrag ist in einer eigenen Zeile dargestellt. Der erste Wert ist das Ende des Items in Sekunden. Der zweite hat keine Bedeutung. Das dritte Feld ist der Name des Items gefolgt von einem Strichpunkt. Im vierten Feld steht die Adresse des Items. Sie muß in jedem Strom eindeutig sein. Danach folgen noch einige Attribut-Werte Paare, die jeweils nur durch ein Leerzeichen getrennt sind. Die Felder danach sind Relationen (Verweise). Sie beginnen jeweils mit einem Semikolon, gefolgt vom Name des Stroms auf den sie Verweisen. Dahinter steht eine Liste von Item-Adressen, die auf die Einträge in den entsprechenden Strömen verweisen.

In der obigen Äußerung ist beispielsweise das Wort mit der Adresse `a428` und dem Namen `zweite` im Wortstrom, mit dem Token mit dem Namen `2` und der Adresse `a415` verknüpft.

Um auf die Utterance-Struktur zuzugreifen stellt Festival einige Schemefunktionen zur Verfügung die in [1] Kapitel 13.5 beschrieben sind. Hier nur ein kurze Beispieldialog:

```
festival> (set! utter (Utterance Text "der 2. April"))
#<UTT 0x87921c0>
festival> (utt.synth utter)
#<UTT 0x87921c0>
festival> (utt.streamnames utter)
(IForm Token Word Phrase Syllable Segment IntEvent Target Wave)
festival> (utt.streamitem.feat utter (utt.stream.tail utter 'Word) "pos")
"N"
festival> (utt.stream.create utter 'Morph)
#<UTT 0x87921c0>
festival> (utt.streamnames utter)
(IForm Token Word Phrase Syllable Segment IntEvent Target Wave Morph)
festival>
```

Zunächst wird mit der Funktion `Utterance` eine neue Äußerung mit dem Typ `Text` erzeugt und der Variablen `utter` zugeordnet. Festival Antwortet mit `#<UTT 0x87921c0>`, was eine interne Variablenbezeichnung für `utter` ist. Mit der Funktion `utt.synth` wird die Äußerung nun synthetisiert, d.h. die Synthese-Module werden in der, im nächsten Abschnitt beschriebenen, Weise auf die Utterance-Struktur angewendet. Prinzipiell könnten die einzelnen Module auch "von Hand" aufgerufen werden in dem nacheinander `(Initialize utter)`, `(Text utter)`, `(Token_POS utter)` `(Token utter)` usw, aufgerufen werden.

Durch Aufruf von `utt.streamnames` wird das System dazu veranlasst, die in der Utterance-Struktur `utter` vorhandenen Ströme zu benennen. Wie man sieht, sind durch den Aufruf von `utt.synth` von den einzelnen Modulen bereits alle Ströme angelegt worden (siehe auch Abblidung 3).

Mit der Funktion `utt.stream.tail` wird nun der letzte Eintrag (Item) aus dem `Word`strom extrahiert ("April") und aus diesem, durch die Funktion `utt.streamitem.feat`, das Part-of-Speech Merkmal `pos` ausgegeben in diesem Fall ein "N" für Noun.

Im nächsten Schritt wird durch `utt.stream.create` ein neuer Strom in die Struktur eingefügt und danach noch einmal Kontrolliert ob dieser nun auch vorhanden ist.

Mit Hilfe der Funktion (`utt.save UTT FILENAME`) kann der Inahlt einer Utterance-Struktur in einem Xlabel ähnliche Format in einer Datei gespeicher werden. Wird als `FILENAME` ``-'' angegeben, wird die Datenstruktur auf dem Bildschirm ausgegeben. Durch Eingabe von (`utt.save utter ``-''`) würde man wiederum die Ausgabe der gesamten Utterance-Struktur wie oben bekommen.

2.3 Die Module in Festival

Wärend die Utterance-Struktur und die verschiedenen Schnittstellen das Grundgerüst für die TTS-Synthese bilden, wird die eigentliche Umsetzung in den einzelnen Modulen gemacht. Module können in C++, oder in Scheme programmiert sein.

Der modulare Aufbau unterstützt das Software Engeneering und ermöglicht es, relativ einfach neue Module einzufügen, bestehende aufzuspalten oder die Reihenfolge ihres Aufrufs zu verändern. Festival bildet damit eine gute Plattform, um neue Entwicklungen in den einzelnen Gebieten der Sprachsynhese in einem Gesamtsystem zu erproben.

Welche Module in welcher Reihenfolge aufgerufen werden wird durch den Äußerungstyp (*utterance type*) bestimmt. In der Datei `lib/synthesis.scm` sind einige grundlegende `UttTypes` definiert. Zu jedem Typ wird bei der definitiom mit Hilfe der Funktion `defUttType` eine Liste von Modulen zugeordnet. Aufgrund dieser Liste entscheidet die Hauptroutiene zur Synthese

(`utt.synth`), welche Module auf die zu bearbeitende Äußerung (*Utterance*) angewendet werden sollen.

Um eine Überblick zu bekommen, wird im folgenden beschrieben welche Module für den Äußerungstyp aufgerufen werden und was in den einzelnen Modulen gemacht wird. Die Abfolge der aufgerufenen Module und deren Zusammenspiel mit der Utterance Datenstruktur ist in Abbildung 3 dargestellt. In den nächsten Abschnitten wird die Funktionsweise der einzelnen Module, wie sie zur Zeit für das Deutsche existieren, beschrieben.

In der Version 1.2 von Festival sind noch einige neue Module hinzugekommen. Da sie in der deutsche Version bisher nicht benutzt werden (bis auf `Token_POS`), sind sie in der nachfolgenden Aufstellung nicht beschrieben. Die aktuelle Liste der Module, die für die einzelnen Äußerungstypen aufgerufen werden, können der Datei `lib/synthesis.scm` entnommen werden.

Eine Instanz der Utterance-Klasse durchläuft nacheinander die verschiedenen Module (vgl. Abbildung 3). Jedes Modul kann auf die Informationen in den vorhandenen Strömen zugreifen, kann neue Ströme hinzufügen oder bereits Vorhandene verändern. Es erhält also als Eingabe eine Liste von Strömen und als Ausgabe eine gegebenenfalls veränderte Liste. Die Utterance-Struktur bildet gewissermaßen die Schnittstelle zwischen den einzelnen Modulen.

So werden sequentiell alle Moduel durchlaufen und in der Utterance-Struktur sukzesiven alle Informationen akkumuliert, welche als Grundlage für die abschließende Synthese gebraucht werden.

Module werden in der Form `(MODULNAME UTT)` aufgerufen. `MODULANME` kann z.B. `Initialize, Text, Token, Phrasify` usw. sein.

Des weiteren können verschiedene Parameter und Variablen für die einzelnen

Module gesetzt werden. Beispiele finden sich in der Datei `lib/german/german_voices.scm`, wo für die einzelnen Stimmen unterschiedliche Parameter gestzt werden.

2.3.1 Initialisierung

Die Initialisierung muß von jeder Äußerung durchlaufen werden. Sie initialisiert die Utterance und lädt die von Anfang an benötigten Ströme aus der Eingabe. In der momentanen Version ist dies nur der `Init`strom, der den rohen Text der Äußerung enthält.

2.3.2 Tokenisierung

Der erste Schritt, um rohen Text zu synthetisieren, ist die Tokenisierung. Ein *Token* in Festival ist ein Atom, welches durch "Whitespaces" vom Rest des Textes abgetrennt ist. Wenn Interpunktionszeichen für eine Sprache definiert sind, werden diese vom Token abgetrennt, und als Merkmal (feature) dieses Tokens im Token-Strom gehalten. Mit `(utt.stream.feat utt token ''punc'')` bzw. `(utt.stream.feat utt token ''whitespace'')` kann darauf zugegriffen werden. Interpunktion und Whitespaces sind in der Datei `lib/token.scm` folgendermaßen definiert:

```
(defvar token.punctuation "\"'`.,:;!?(){}[]")
(defvar token.prepunctuation "\"'`({[")
(defvar token.whitespace " \t\n\r")
```

Für das Deutsche wurde diese Art der Tokenisierung zunächst übernommen und lediglich die prepunctuation-Menge um den Bindestrich `'-'` erweitert. Bei Zusammensetzungen, wie Beispielsweise *610-Mark-Jobs*, ist diese Lösung

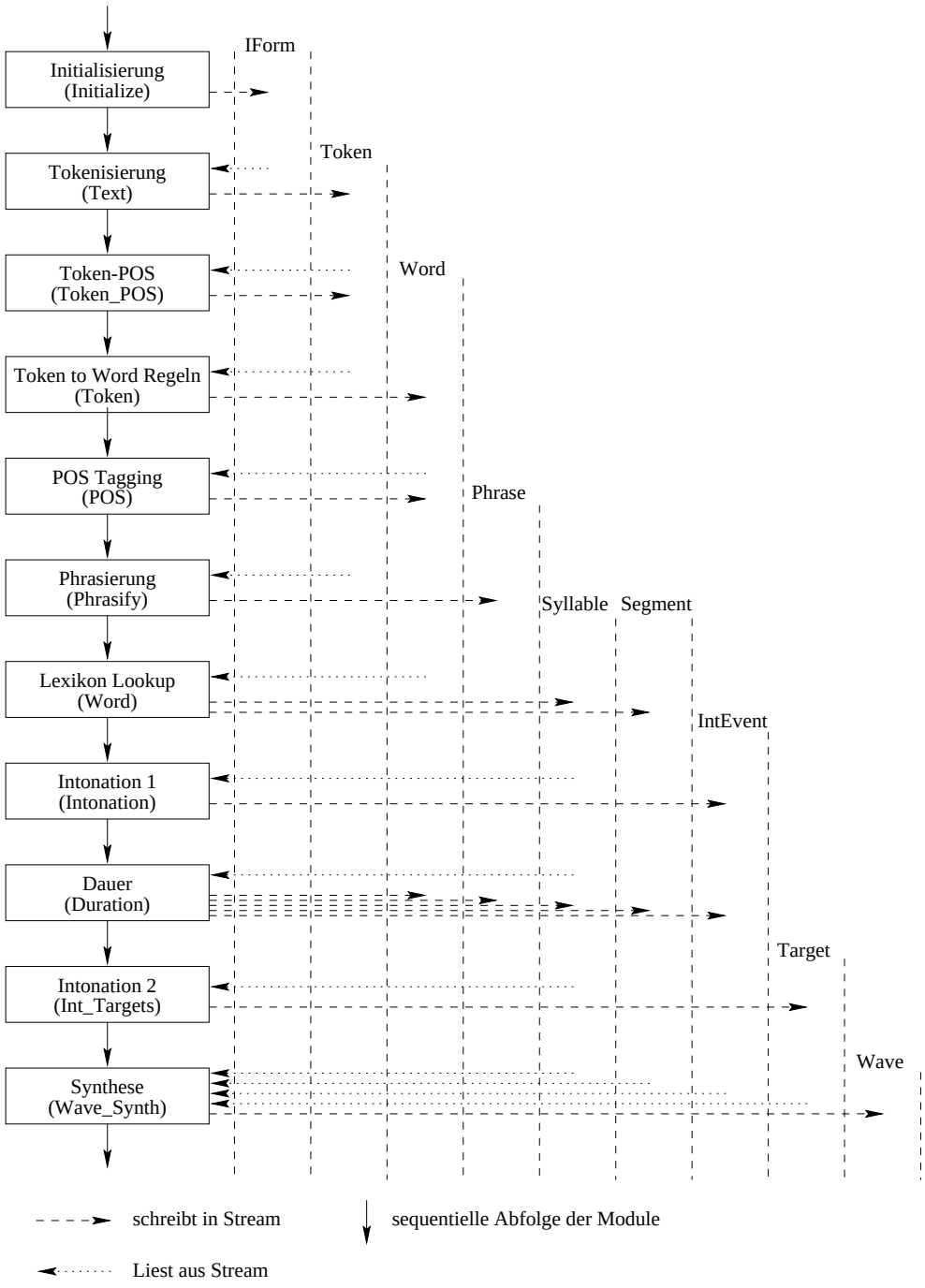

Abbildung 3: Die Module des Festival TTS-Systems und deren Zugriffe auf die Utterance-Struktur (unvollständig)

allerdings oft unbefriedigend. In zukünftigen Versionen könnte man sich überlegen, ob man eine etwas Aufwendigere Tokenisierung, z.B. mit Hilfe von GNU Lex, durchführen sollte.

2.3.3 Token POS

Ab der Festival Version 1.2 existiert ein neues Modul zur Disambiguierung von Homographen. Mit Hilfe von sogenannten *CART-lists* (Liste von Entscheidungsbäumen), wird jedem Token im Tokenstrom ein sogenanntes `token_pos` zugeordnet. Diese Information kann dann bei den Token-to-Word Regeln benutzt werden. Bei der deutsche Version wird dieses Modul z.B. dazu verwendet, Ordinalzahlen von Kardinalzahlen zu unterscheiden. Die genauere Funktionsweise ist in [1] Kapitel *homograph disambguation* beschrieben.

Die Part-of-Speech Informationen werden als Merkmal (featur) in den Tokenstrom geschrieben. Sie können mit Hilfe der Funktion `utt.streamitem.feat` wieder ausgelesen werden.

2.3.4 Token to Word Regeln

In diesem Modul werden Token aus dem Tokenstrom gelesen und in eventuell abgewandelter Form in den Wortstrom geschrieben. Dieses Modul ist hauptsächlicher Gegenstand dieser Studienarbeit. Insbesondere müssen Zahlen, Abkürzungen und Sonderzeichen expandiert werden.

Der Tokenstrom `Sie bekamen 12% der Stimmen.` wird beispielsweise in den Wortstrom `Sie bekamen zwölf Prozent der Stimmen.` übersetzt.

2.3.5 POS Tagging

Der *Part-of-Speech-Tagger* bestimmt, mit Hife eines probabilistischen Modells, verschieden Tags (z.B. Wortart) für die Wörter aus dem, im vorangegangenen Modul erzeugten, Wortstrom. Die POS-Informationen werden direkt in den Wortstrom geschrieben.

In der deutschen Version ist dieses Modul bis jetzt ungenutzt (die Variable `pos_lex_name` ist auf `NIL` gesetzt). Eventuell kann an dieser Stelle der am IMS entwickelte POS-Tagger von Helmut Schmidt (u.a.) eingesetzt werden. Im Moment ist allerding geplant eine syntaktische Analyse, welche auf der Morphologie DMOR aufbaut, einzusetzen (vgl. Abschnitt 5.2.4). In diesem Fall wäre ein POS-Tagger obsolet.

2.3.6 Phrasierung

Im Modul `Phrasify` werden die Pausen zwischen zwei Phrasen bestimmt. Dies geschiet in der deutschen Version bis jetzt mit folgendem Entscheidungsbaum (classification tree)[3]:

```
(set! german_phrase_cart_tree
'((lisp_german_end_punc matches ".*[?.,:;].*")
 ((B))
 ((n.name is 0)
  ((B))
  ((lisp_german_end_ged  is "-")
   ((B))
   ((NB))))))
```

Dieser einfache Entscheidungsbaum besagt, daß hinter diversen Satzzeichen, am Ende einer Äußerung (`(n.name is 0)`) oder wenn das nächste Token ein

[3]Die Funktionsweise von Entscheidungsbäumen ist in Abschnitt 2.4.1 und in [1] Kapitel 23.2 kurz beschrieben

Gedangkenstrich ist, eine Pause (B) in den Phrasestrom eingefügt wird. In allen anderen Fällen wird keine Pause (NB) gemacht.

Des weiteren bietet Festival die Möglichkeit, Sprechpausen mittels eine probabilistischen Modells (N-Gramm Modell der umgebenden Worte) zu bestimmen. Dies ist in [1] Kapitel *Phrase breaks* genauer beschrieben.

Die Pauseninformation wird in den, durch dieses Modul neu hinzugefügten Strom, `Phrase` geschrieben.

2.3.7 Lexikon-Lookup

Im Modul `Word` findet die Graphem-Phonem Konversion (vgl. Abblidung 2) statt. Dieser Vorgang teilt sich in drei Schritte: zuerst wird versucht, ein Wort in der *Addenda* zu finden. Die Addenda ist ein, vom Beutzer selbst zu erstellender Zusatz zum Lexikon, welcher typischer Weise wenige, anwedungsspezifische Einträge (Zum Beispiel den Namen des Bemutzers) umfaßt. Zur Addenda können mit Hilfe der Lisp-Funktion `lex.add.entry` eigene Einträge hinzugefügt werden. Ein Beispiel hierfür ist in der Datei `lib/german/german_lexicon` zu finden. Wird das Wort in der Addenda nicht gefunden, wird es im Lexikon nachgeschlagen (zur Zeit wird das deutsche Vollformlexikon von *Celex* (vgl. [19] und [25]) benutzt, schlägt dies fehl, werden sogenannte *Letter-to-Sound* Regeln verwendet, um das Wort in seine Lautschrift zu übersetzen.

Da diese Regeln oft unbefriedigende Resultate liefern, muß darauf geachtet werden, daß die meisten Wörter im Lexikon vorhanden sind. Deshalb ist es sinnvoll, in derivation-, flexions- und kompositionsreichen Sprachen wie dem Deutschen, eine morphologische Komponente einzusetzen. Wie dies gesche-

hen könnte wird in Kapitel 5.2 beschrieben.

Zusätzliche Ströme: Syllable, Segment

2.3.8 Intonation 1

Im allgemeinen wird die Intonation in zwei Schritten bestimmt:

1. Bestimmung der Wortakzente auf Silbenbasis
2. Bestimmung der F0-Intonationsziele. Diese müssen nach der Dauer bestimmte werden (siehe 2.3.10)

Zusätzlicher Strom: IntEvent

2.3.9 Dauer

Im `Duration` Modul wird die Dauer der einzelnen Segmente bestimmt.

2.3.10 Intonation 2

Die Bestimmung der Intonationsziele ist in [1] Kapietel 17 beschrieben.

Zusätzlicher Strom: Target

2.3.11 Synthese

Das Modul `Wave_Synth` erledigt die Umsetzung vom symbolischen in den numerischen Bereich. Es erzeugt aus den Informationen der vorangegangenen Modulen das eigentliche Sprachsignal, welches dann, mit Hilfe entsprechender Hardware (DA-Wandler, Lautsprecher ...), hörbar gemacht werden kann.

Im Moment wird hier der Diphonsyntsizer aus MBROLA Projekt eingesetzt (vgl. [24]). Geplant ist auch ein Synthesemodul, das mit dem *unit selection* Verfahren arbeitet (vgl. [16] und [23])

2.4 Die Werkzeuge von Festival

Außer dem Grundgerüst und den Modulen zur TTS-Synthese, stellt Festival noch mit Hilfe der *Edinburgh Speech Tools Library* eine Reihe von Werkzeugen zur Verfügung, die in der Sprachverarbeitung immer wieder gebraucht werden. Dies sind insbesondere *reguläre Ausdrücke, CART trees, Ngramms, Viterbi Dekoder* und *lineare regression*. Die einzelnen Werkzeuge sind in [1] Kapitel 23 beschrieben.

In den nächsten beiden Abschnitten werden nur die Erweiterungen beschrieben, die noch nicht dokumentiert (bei den CART trees), bzw. wärend dieser Studienarbeit neu hinzugekommen sind (Patternmatcher).

2.4.1 Erweiterung der CART trees

CART trees steht für Classification and Regression trees. In der Systemdokumentation von Festival [1] wird die Syntax einer Frage folgendermaßen angegeben:

```
QUESTION ::= ( FEATURE in LIST )
QUESTION ::= ( FEATURE is STRVALUE )
QUESTION ::= ( FEATURE = NUMVALUE )
QUESTION ::= ( FEATURE > NUMVALUE )
QUESTION ::= ( FEATURE < NUMVALUE )
QUESTION ::= ( FEATURE matches REGEX )
```

`FEATURE` ist ein Merkmal wie es in [1] Kapietel 13.6 beschrieben ist. In der Version 1.2 von Festival können allerdings auch Lispfunktionen an Stelle von

FEATURE eingesetzt werden. Ein Beispiel hierfür ist der `german_phrase_cart_tree` (vgl. 2.3.6). Die Lispfunktion muß lediglich zwei Parameter besitzen: eine Utterance Struktur und einen Streamitem. Um Festival anzuzeigen, daß es sich nicht um eine FEATURE, sondern um eine Lispfunktion handelt, wird vor den Funktionsnamen ein `lisp_` geschrieben.

Außerdem gibt es noch eine Erweiterung, die den rechten Operator einer Frage betreffen (LIST, STRVALUE ...) und die in der Version 1.2 von Festival noch nicht eingebaut ist. Diese Erweiterung soll aber spätestens in der Version 1.3 fester Bestandteil des Systems sein. Bis jetzt ist er als Patch in die Datei `src/arch/festival/wagon_interp.cc` eingebaut. Und zwar ist dort die Funktion `wagon_ask` durch eine neue Version ersetzt worden. Die Änderungen stammen direkt von Allan Black und werden in der Version 1.3 höchstwahrscheinlich in genau dieser Form integriert sein. Die Änderung macht es möglich auf der rechten Seite, statt den angegebenen Werten, auch beliebige Lispausdrücke einzusetzen. Durch die Voranstellung eiens Kommas wird dem System angezeigt, daß es den nachfolgenden Lispausdruck auswerten soll. Ein Beispiel hierfür zeigen die `german_token_pos_cart_trees` in 3.2

2.4.2 Erweiterung des Festival-Regex-Tools

Festival-Regex: *string-matches* Reguläre Ausdrücke sind ein formales Mittel um reguläre Sprachen zu beschreiben. Mit ihrer Hilfe kann ein Muster angegeben werden, welches eine Klasse von Zeichenketten beschreibt. Reguläre Ausdrücke sind weit verbreitet in Scriptsprachen wie Perl, awk, Emacs-Lisp u.s.w.

Festival stellt der BenutzerIn eine Scheme-Schnittstelle (`string-matches`) zur Verwendung von regulären Ausdrücken, ähnlich der GNU libg++ `Regex-`

Klasse, zur Vergfügung. Implementierung und Anwendung, insbesondere die Form der regulären Ausdrücke, sind in [1] Kapitel 23.1 genauer beschrieben. Die Scheme-Funktion `string-matches` nimmt einen String und einen regulären Ausdruck und gibt bei Übereinstimmung `t` zurück.

Die Erweiterung: *pattern-matches* Oft wäre allerdings wünschenswert, daß die Funktion nicht nur zurückgibt, daß der reguläre Ausdruck den String matcht, sonder, wie beispielsweise in PERL (vgl. [7] Seite 24 "Pattern Matching"), die "gematchten" Teilstrings an Variablen bindet, also eine echtes Pattern Matching, durchführen. Da Pattern Matching im allgemeine Fall mit erheblichem Zeitaufwand verbunden ist (vgl. [8] Kapitel 5.2), wird hier nur eine sehr eingeschränkte, aber für die meisten Anwendungen ausreichende Form implementiert: Es wird vom Beginn des Strings an immer nur der längste Match berücksichtigt. Die neue Scheme-Funktion heißt `pattern-matches` und nimmt, wie `string-matches` einen String und ein Pattern. Das Pattern hat allerdings eine leicht abgewandelte Form: Das Pattern ist zunächst aufgebaut wie in [1] Kapitel 23.1 "Regular expressions" beschrieben:

`.`	matcht jedes Zeichen.
`$`	matcht das Ende eines Strings.
`^`	matcht den Anfang eines Strings.
`X*`	matcht kein, oder mehrere Vorkommen von X; X kann ein Zeichen, ein Bereich oder ein geklammerter Ausdruck sein.
`X+`	matcht ein oder mehrere Vorkommen von X; X kann ein Zeichen, ein Bereich oder ein geklammerter Ausdruck sein.

`X?` matcht ein oder kein Vorkommen von X; X kann ein Zeichen, ein Bereich oder ein geklammerter Ausdruck sein.

`[...]` ein Bereich matcht einen der Werte in den Klammern. Der Bereichoperator "-" erlaubt die Spezifiaktion eines Bereichs, z.B. matcht `[a-zäöüß]` alle Kleinbuchstaben inklusive deutscher Umlaute. Ist das erste Zeiche in der Klammer ein "^" so werden alle Zeichen, außer den in den Klammern angegebenen, gematcht. Soll das "-" im Bereich vorkommen muß es an die erste Stelle gesetzt werden.

`\\(...\\)` so geklammerte Ausdrücke werden wie einzelne Zeichen behandelt. Auf diese Weise können '*', '+', '?' etc auf mehr als nur einzelne Zeichen angewendet werden.

`X\\|Y` matcht entweder X oder Y. X oder Y können ein Zeichen, ein Bereich oder ein geklammerter Ausdruck sein.

Es ist zu beachten, daß normalerweise nur ein Escape-Backslash vor einem Zeichen benötigt wird. Reguläre Ausdrücke werden aber meist in C++ oder Scheme-Strings benutzt, in denen der Backslash selbst durch einen Backslash geschützt werden muß. Deshalb werden meist zwei Backslashes benötigt.

Für die neue Funktion `(pattern-matches string pattern)` kommt eine weiter Klammerung hinzu:

{...} für jeden in geschweiften Klammern stehende Ausdruck wird eine neue Variable generiert, an welche der Teil des Strings gebunden wird, der den regulären Ausdruck in den Klammern matcht. Die Variablen haben von links beginnend die Namen #1, #2, #3, usw.

Zum bessern Verständnis hier ein Beispiel:

`(pattern-matches ``9pfünder'' ``{[0-9]+}{[a-zäöüß]+}'')` liefert `t` und bindet "9" an #1 und "pfünder" an #2

3 Die deutsche Textvorverarbeitung in Festival

Wie bereits beschrieben, müssen in einem TTS-System, welches rohen Text verarbeiten kann, folgenede Besonderheiten behandelt werden:

- Formatierungen und Gliederungen
- Zahlen
- Abkürzungen
- Sonderzeichen
- Interpunktion
- Worte in besonderer Schreibweise

Um dies zu bewerkstelligen, wird vor die linguistische Verarbeitung eine Textvorverarbeitung geschaltet.

Die Expansion von Ordinalzahlen und Abkürzungen wird in manchen TTS-Systemen (z.B. [15]) in die syntaktische Analyse integriert. Dafür sprich, daß deren Übersetzung in die volle phonetische oder ortographische Form meist vom syntaktischen Kontext abhängt. Ein Beispiel hierfür ist:

"am 7. Tag" $\rightarrow$ "am *siebten* Tag" (dativ)
"der 7. Tag" $\rightarrow$ "der *siebte* Tag" (nominativ)

Wegen der besseren Übersichtlichkeit, dem Erhalt der Modularität und nicht zuletzt wegen der noch fehlenden syntaktischen Analyse, wird diese Möglichkeit in der vorliegenden Arbeit nicht erwogen. Wenn in späteren Versionen eine syntaktische Analyse gemacht wird, wäre zu überlegen, ob die Textvorverarbeitung eventuell in zwei Teile aufgespalten wird. Dieser Vorgang ist in Abschnitt 5.1 beschriebenen.

Die Textvorverarbeitung wird in Festival im wesentlichen durch drei Module realisiert: `Text, Token_POS` und `Token`. Im Modul `Text` wird der eingegebene Text (im Strom `IForm`) tokenisiert und in den Strom `Token` geschrieben. Dieser Vorgang ist bereits im Abschnitt 2.3.2 beschrieben. Das Modul `Token_POS` dient der disambiguierung von Homographen (vgl. Abschnitt 2.3.3). Es ordnet einigen Token ein sogenanntes `Token_POS` zu. Es wird in mit in den `Token`strom geschrieben . Die Hauptarbeit erledigt allerdings das Modul `Token`. Mit Hilfe der Funktion `(token_to_words UTT TOKEN NAME)` wird dort bestimmt, ob und wie ein Token übersetzt und in den `Word`strom geschrieben wird. `token_to_words` wählt je nach Sprache, die entsprechende sprachspezifische Funktion. Für das Deutsche ist dies die Funktion `(german_token_to_words UTT TOKEN NAME)`.

Die meisten Token entsprechen einem Wort und können unbearbeitet vom Tokenstrom in den `Wort`strom übernommen werden. Andere Token, wie z.B.

Abkürzungen oder Zahlen, müssen zunächst übersetzt werden.

`german_token_to_words` ist vollständig in Scheme implementiert und findet sich in der Datei `lib/german/german_token_to_words.scm`. Sie benutzt diverse Hilfsfunktione, die zum größten Teil in der Datei `lib/german/german_token_to_words_tools.scm` definiert sind. Sie liefert als Ergebnis eine Liste von Worten, welche als Übersetzung des Tokens in den Wordstrom geschrieben werden. Was in den einzelnen Strömen nach erfolgter Synthese tatsächlich steht, kann mit Hilfe der Funktion `test_show` kontrolliert werden. Diese Funktion ist in `lib/german/debugtools.scm` definiert.

3.1 Aufsplittung von Zusammensetzungen

Repräsentiert ein Token eine Zusammensetzung, wie z.B. *610-Mark-Jobs*, muß dieses zunächst in seine Bestandteile aufgesplittet werden (*610*, *Mark* und *Jobs*). Diese müssen dann einzeln von der Funktion `german_token_to_words` bearbeitet werden.

3.2 Expansion von Zahlen

Eine der wichtigsten Aufgaben der Tetxtvorverarbeitung ist die Umwandlung von Ziffernfolgen in ihren Wortlaut. Ziffernfolgen kommen dabei in verschieden Schreibweisen vor und werden, teilweise abhängig vom Kontext, unterschiedlich ausgesprochen. Grob lassen sich folgende Zahlenformate unterscheiden:[4]

[4]die Reihenfolge der Aufzählung entsprich in etwa der Reihenfolge der Behandlung in `lib/german/german_token_to_words.scm`

- Brüche wie *23/64*
- Verhältniszahlen wie *2:0*
- Telefonnummern wie *0711/123321*
- Zusammensetzungen wie *10fach*, *16pfünder*, *Jäger90*, *B52* usw.
- Jahreszahlen wie *1997*
- Datumsangaben wie *13-05-97*
- Uhrzeiten wie *12.30 Uhr* oder *13:55h*
- Geldbeträge wie *17,59 DM*
- Dezimalbrüche wie *21,45*
- Ordinalzahlen wie in *der 13. Teilnehmer*
- Kardinalzahlen wie *3465*
- römische Zahlen wie *XIV*

Für die korrekte Umsetzung muß zunächst herausgefunden werden, um welche Art von Ziffernfolgen es sich handelt. Dies geschieht mit Hilfe der Funktion `german_numbers_to_words` in der Datei `lib/german/german_token_to_words.scm`. Zahlen müssen kontextabhängig umgesetzt werden, da sie oft bei gleicher Schreibweise unterschiedlich ausgesprochen werden. Ein Beispiel dafür ist "1968.". Diese Ziffernfolge mit anschließendem Punkt kann dreierlei Bedeutungen haben und wird je nach Bedeutung unterschiedlich ausgesprochen:

1. *... im Jahre 1968.* wird zu *neunzehn-hundert-acht-und-sechzig.*

2. *... am ende waren es 1968.* wird zu *ein-tausend-neun-hundert-acht-und-sechzig.*

3. *... er war der 1968. Besucher* wird zu *ein-tausend-neun-hundert-acht-und-sechzigste*

Im letzten Fall muß zusätzlich aus dem Kotext die Endung bestimmt werden. Festival stellt ein Werkzeug zur Verfügung, um solche Homographen aufzulösen.[5] Mit Hilfe von sogennannten *Entscheidungslisten* (CART-Lists) kann jedem Token ein *Part-of-Speech Tag* (`Token_POS`) zugeordnet werden. Ein Beispiel für eine solche CART-Liste ist die Variable `german_token_pos_cart_trees`

```
(set! german_token_pos_cart_trees
  '(
    ;;  Format is (Regex Tree)
    ("[0-9]+" ;;digits only -> cardinal, ordinal or year?
     ((punc matches "\\..*") ;; number followed by a "."?
      ((n.lisp_ger_token_pos_guess is sos)
       ;; nächstes Token Satzanfang -> Cardinal
       ((p.name in lisp_german_year_indicator)
        ;; Jahr oder Card?
        ((year))
        ((p.lisp_ger_token_pos_guess is month)
         ((year))
         ((pp.name is als)
          ((year))
          ((cardinal)))))
       ((n.name is 0) ;; end of utterance?
        ((p.name in ,german_year_indicator)
         ;; Jahr oder Cardinal?
         ((year))
         ((p.lisp_ger_token_pos_guess is month)
          ((year))
          ((pp.name is als)
           ((year))
```

[5] vgl. [1] Kapitel 14.3 “Homograph Disambiguation”

```
          ((cardinal)))))
        ((p.lisp_ger_token_pos_guess is month)
         ((year))
         ((p.name in (Jahr Jahre Ende))
          ((year))
          ((ordinal))))))
      ((p.name in ,german_year_indicator)
       ;; Jahr oder Cardinal?
       ((year))
       ((p.lisp_ger_token_pos_guess is month)
        ((year))
        ((pp.name is als)
          ((year))
          ((cardinal)))))))

    ("[1-9][0-9]*\\,[0-9][0-9]"    ;; potentieller Kandidat
     ((n.name in ,german_currency) ;; für Geldbetrag
      ((money))
      ((dezimalbruch))))
))
```

mit Ihrer Hilfe wird außer der oben genannten Mehrdeutigkeit, auch die zwischen normalen Dezimalzahlen und Währungsangaben ausgeräumt.

Danach können sie mit Hilfsfunktionen umgesetzt werden. Diese Hilfsfunktionen sind zum Teil in C (in der Datei `src/modules/german_text/german_text.cc`) und zum anderen Teil in Scheme (in der Datei `lib/german/german_token_to_words_tools.scm`) implementiert.

In den folgenden Abschnitten wird die Umsetzung der o.g. Ziffernfolgen in die entsprechende orthographische Form erläutert.

3.2.1 Brüche

Brüche haben die Form[6] ZÄHLER/NENNER, wobei Zähler und Nenner jeweils Kardinalzahlen sind. Die Erkennung eines Bruches erfolgt mit Hilfe der Funktion `pattern-matches` (vgl. 2.4.2). Die Umsetzung des Zähler übernimmt die Funktion `german_parse_cardinal` (vgl. 3.6), der Bruchstrich wird nicht gesprochen und der Zähler wird mit Hilfe der Funktion `german_parse_fractal` (vgl. 3.6) übersetzt.

Beim Test hat sich allerdings herausgestellt, daß die Umsetzung auf diese Weise nicht sinnvoll ist, da das Format KARDINAL/KARDINAL oft dazu verwendet wird, um z.B. Jahreszahlen anzugeben (z.B. *im WS 97/98*).

Um diese Homographen zu unterscheiden, müßten ähnliche Mechanismen eingesetzt werden, wie bei der Unterscheidung Kardinalzahl/Jahreszahl. Dies ist bis jetzt nicht realisiert. Die Umsetzung von Brüchen wurde deshalb herausgenommen. Token in entsprechendem Format werden nun als zwei Kardinalzahlen gesprochen.

3.2.2 Verhältniszahlen

Verhältniszahlen, wie sie beispielsweise in Sportnachrichten häufig vorkommen,haben die Form KARDINAL:KARDINAL. Die Erkennung wird von `string-matches` (vgl. 2.4.2) erledigt. Die Umsetzung der Kardinalzahlen übernimmt wiederum `german_parse_cardinal` und dazwischen wird ein 'zu' eingefügt.

[6] *haben die Form* bedeutet hier jeweils: wenn sie nicht diese Form haben, werden sie nicht korrekt klassifiziert und damit nicht korrekt umgesetzt

3.2.3 Telefonnummern

Telefonnummern haben das gleiche Format wie Brüche, nur daß sie mit einer '0' beginnen und statt des Schrägstriches auch ein Bindestrich stehen kann. Telefonnummern werden Ziffernweise vorgelesen. Die Erkennung wird wiederum mit `string-matches` durchgeführt, die Expansion erfolgt mit `german_parse_charlist` (vgl. 3.6).

3.2.4 Zusammensetzungen

Zusammensetzungen wie *Jäger*90 und 16*jährig* werden mit der Hilfsfunktion `pattern-matches` erkannt. Die Zahl wird mit `german_parse_cardinal` übersetzt, das enthaltene Wort wird einfach vorn bzw. hinten angehängt.

3.2.5 Jahreszahlen

Jahreszahlen zwischen 1100 und 1999 werden als 'xx Hundert yy' gesprochen, wobei xx die ersten beiden Ziffern und yy die letzten beiden Ziffern bezeichnet. Ob eine Cardinalzahl eventuell eine Jahreszahl ist wird bereits mit Hilfe von `german_token_pos_cart_trees` (siehe Abschnitt 3.2) entschieden. Da diese Unterscheidung (noch) recht unzuverlässig ist, werden alle Kardinalzahlen, die im oben angegebenen Intervall liegen, wie eine Jahreszahl gesprochen.

3.2.6 Datumsangaben

Eine Datumsangabe hat die Form TAG.MONAT.JAHR. Statt der Punkte sind auch Bindestriche erlaubt. JAHR kann vierstellig oder zweistellig

geschrieben sein oder auch ganz weggelassen werden. Tag und Monat werden als Ordinalzahlen gesprochen, JAHR als Kardinalzahl. Die Erkennung wird wiederum mit Hilfe von `pattern-matches` vorgenommen, eine Bereichsprüfung der Zahlen findet nicht statt. Zur Übersetzung werden die Hilfsfunktionen `german_parse_cardinal` und `german_parse_ordinal` verwendet. Zur Bestimmung der Endungen bei Ordinalzahlen dient der `german_ordinal_prediction_tree` (siehe auch 3.2.10 und 3.6).

3.2.7 Uhrzeiten

Uhrzeiten haben die Form STUNDEN.MINUTEN, gefolgt von dem Token 'Uhr' oder 'h'. Statt des Punktes ist auch ein Doppelpunkt erlaubt. Ausgesprochen werden sie STUNDEN 'Uhr' MINUTEN. Die Erkennung des oben angegebenen Formats geschieht wiederum durch `pattern-matches` mit zusätzlicher Überprüfung des nachfolgenden Tokens. Zusätzlich wäre an dieser Stelle noch eine Überprüfung des Wertebereichs ($0 \leq$STUNDEN≤ 24 und $0 \leq$MINUTEN≤ 59) möglich. Die Umsetzung der Stunden und Minuten wird wiederum durch `german_parse_cardinal` realisiert. Zusätzlich muß darauf geachtet werden, daß das Wort 'Uhr' nicht zweimal ausgesprochen wird. Deshalb wird jedes Token darauf untersucht, ob das vorherige Token als Uhrzeit erkannt wurde. Dies geschieht durch die Abfrage des Mekmals `Token_POS`. Ist dies der Fall wird das entsprechende Token nicht in den Wortstrom geschrieben.

3.2.8 Geldbeträge

Geldbeträge haben die Form KARDINAL,KARDINAL und das nächste Token bezeichnet einen Einheit (z.B. DM, sfr ...). Ausgesprochen wird ein Geld-

betrag, indem die entsprechende Einheit zwischen die beiden Kardinalzahlen eingefügt wird. *1,34 DM* wird beispielsweise als *eine Mark vier und dreißig* gesprochen.

Die Erkennung von Geldbeträgen wird bereits im Modul `Token _POS` vorgenommen. In `greman_numbers_to_words` muß dann nur geprüft werden, ob das Merkmal `Token_POS` des Tokens entsprechend gesetzt ist. Die Übersetzung der Zahlen übernimmt wiederum `german_parse_cardinal`. Zusätzlich wird die ortographische Form der Währungseinheit durch die Funktion `german_fetch_currency` ermittelt. Wie bei der Uhrzeit, wird muß auch hier gewärleistet werden, daß das nachfolgende Token (die Währunseinheit) nicht noch zusätzlich ausgesprochen wird.

3.2.9 Dezimalbrüche

Alle Ziffernfolegen die bis jetzt noch keinem Format entsprochen haben und die ein Komma enthalten (KARDINAL,KARDINAL), sind dezimalbrüche und werden entsprechend übersetzt. Die Zahl vor dem Komma mit Hilfe von `german_parse_cardinal`, das Komma wird als solches gesprochen und die Nachkommazahl wird mit `german_parse_charlist` in einzelne Ziffern zerlegt, die einzeln ausgesprochen werden.

3.2.10 Ordinalzahlen

Ordinalzahlen haben das gleich Format wie Kardinalzahlen (siehe dort), nur daß sie immer von einem Punkt gefolgt sind. Das Problem dabei ist, daß Kardinalzahlen, welche am Satzende stehen, ebenfalls durch einen Punkt beendet werden. Zur korrekten Unterscheidung ist deshalb eine Erkennung des

Satzendes von besonderer Bedeutung. Eine aufwendige Methode zur Satzgrenzenbestimmung, wie sie Beispielsweise in [13] beschrieben wird, hätte den Rahmen dieser Studienarbeit gesprengt. Statt dessen wurde hier ein sehr einfaches Verfahren, welches bereits zu recht guten Ergebnissen führt, verwendet[7]: Das Token nach dem Punkt wird mit einer Menge von Worten[8] verglichen, welche nur am Satzanfang großgeschrieben werden. Ist es darin enthalten, ist der Punkt sicher ein Satzendpunkt. Wenn nicht, wird davon ausgegangen, daß der Punkt zu der vorangegangenen Ordinalzahl gehört.[9]

Das zweite große Problem von Ordinalzahlen besteht in der Kontextabhängigkeit der Übersetzung. Je nach Kasus, Genus und Nummerus müssen sie unterschiedlich flektiert werden. Um dieses Problem zu lösen, kann man zwei verschiedene Strategien verfolgen: Entweder man expandiert so weit wie möglich und markiert die Übersetzungen als unvollständig. Nach erfolgter syntaktischer Analyse können dann spätere Module mit den zusätzliche Informationen das Ergebnis vervollständigen. Oder man versucht mit Hilfe von mehr oder weniger brauchbaren Heuristiken die komplette Expansion sofort zu bestimmen. Nicht zuletzt weil noch kein Modul zur syntaktischen Analyse für Festival existiert, wurde in dieser Arbeit der letztere Weg gewählt. Wenn Festival um eine Syntax- und Semantikkomponente erweitert wird, können die entsprechenden Analyseergebnisse gewinnbringend eingesetzt werden.

Momentan werden die Endungen mit dem `german_ordinal_prediction_tree` bestimmt, der in der Datei `lib/german/german_token_to_words_lists` definiert wird.

[7] dieses Verfahren wird auch in [6] beschrieben

[8] diese Menge ist in der Variablen `german_satzanfang` gespeichert, die in der Datei `lib/german/german_token_to_words_lists.scm` definiert ist. Sie kann von der BenutzerIn ergänzt werden.

[9] das selbe Verfahren wird auch bei Abkürzungen mit Punkt verwendet.

Sobald die (hoffentlich) korrekte Endung mit Hilfe dieses Entscheidungsbaums und der von Fesival zur Verfügung gestellten Funktion `wagon` bestimmt wurde, wird die Ordinalzahl mit der Funktion `german_parse_ordinal` expandiert.

3.2.11 Kardinalzahlen

Kardinalzahlen bestehen entweder aus einer geschlossenen Ziffernfolge oder sind durch Punkte oder Leerzeichen in Dreierblöcke gegliedert. Im Falle einer Gliederung müssen sie zunächst (rekursiv) zu einer geschlossenen Ziffernfolge zusammengezogen werden. Ist dies erfolgt, können sie mit der Funktion `german_parse_cardinal` expandiert werden.

3.2.12 römische Zahlen

Römische Zahlen werden zunächst mit Hilfe der Funktion `ger_tok_roman_to_numstring` in eine Folge von arabischen Ziffern übersetzt. Danach werden sie wie Ordinalzahlen übersetzt. Geht der römischen Zahl ein Königs- oder Kaisername voraus, wie z.B. in *Ludwig XIV*, muß außerdem noch ein *der* eingefügt.

3.3 Behandlung von Abkürzungen

Eine weitere zentrale Aufgabe der Textvorverarbeitung ist die Behandlung von Abkürzungen. Die fehlenden verbindlichen Richtlinien zur Bildung von Abkürzungen und deren enorme Anzahl, stellen eine große Herausforderung an die Textvorverarbeitungs-Komponente eines TTS-Systems. Weitere Probleme sind die Mehrdeutigkeiten, die meist nur aus dem semantischen Kon-

text aufgelöst werden können, sowie die Tatsache, daß sie in vielen Fällen flektiert werden müssen.[10]

Im Duden Wörterbuch der Abkürzungen ([12]) werden drei Klassen, die sich in ihrer Schreibweise deutlich gegeneinander abgrenzen, unterschieden:

1. allgemeine Abkürzungen

2. Kurzforman die als Zeichen oder Symbole behandel werden. Diese, vor allem im Bereich Naturwissenschaft, Technik und Wirtschaft, als Zeichen und Symbole (Maß bzw. Maßeinheit, Gewicht, Element usw.) verwendeten Abkürzungen werden ohne Punkt geschrieben. Dies sind z.B. *km, GeV, kWh, MIPS ...*

3. Kurzformen, die im eigentlichen Sinne nicht mehr als Abkürzungen angesehen, sondern Neuwörter betrachtet und im Sprachgebrauch selbstständig verwendet werden. Diese Kurzformen längerer Bezeichnungen oder Kurzbildungen aus bestimmten Bestandteilen mehrerer Wörter einer mehrteiligen Bezeichnung werden ebenfalls ohne Punkt geschrieben. Sie werden meist wie echte Substantive im Text behandelt. Beisbiele hierfür sind *Akku, Unesco, Nato ....*

Die Abkürzungen aus Gruppe zwei und drei sind in Bezug auf ihre Rechtschreibung weitgehend einheitlich ausgerichtet, da ihre Schreibung auf normativen Entscheidungen von Fachgremien beruht.

Die erste Klasse stellt nicht nur die größte, sondern auch eine völlig uneinheitliche dar. Sie wird nach [12] in weitere vier Untergruppen aufgeteilt:

[10] auf die letzten beiden Probleme wird in der Vorliegenden Arbeit nicht eingegangen

1. Einfache Abkürzungen, die in vollem Wortlaut gesprochen werden, erhalten einen Schlußpunkt. Dabei spielt es keine Rolle, ob die Abkürzungen durch Weglassen aller weiteren Buchstaben nach den ersten Buchstaben (im Altertum un Mittelalter *Suspension* genannt) oder durch Zusammenziehen des ersten und letzten Buchstabens (im Altertum und Mittelalter *Kontraktion* genannt) oder durch andere Buchstabenkombinationen gebildet werden (*z.B.; u.ä., i.A.; Nr., Bf., Jg., Dr., frz., med.*).

2. Abkürzungen, die nicht mehr in vollem Wortlaut, sondern als Abkürzung gesprochen werden, erhalten im allgemeinen keinen Punkt: *Kfz, Lkw/LKW, Pkw/PKW, BGB, KG, AG, PDS, VDI.* Ausnahmen bilden hier traditionelle Schreibungen (*Co., Cie.; z. B. V.*) oder Abkürzungen äusländischer Herkunft (*k.o.*).

3. Im deutschen werden Abkürzungen die nur in Versalien geschrieben, aber nicht selbstständig gesprochen werden, meist ohne Punkt(e) geschrieben: *AA (= Auswärtiges Amt), GMD (= Generalmusikdirektor).* Bei Abkürzungen aus dem Ausland werden auserhalb des Deutschen in diesen Fällen noch häufig Punkte gestzt: *R.A.F. (Royal Air Force); U.S.A. (United States of America; B.N. (Banque Nationale).*

4. Eine große Gruppe bildet die Abkürzung zusammengesetzter Wörter: zusammengestzte Wörter, die in ihrern einzelnen Bestandteilen abgekürzt werden, müssen nach Duden-Regel in die Bestandteile aufgelöst und mit Punkt und Bindestrich geschrieben werden. (z.B. *Reg.-Dir.; Vers.-Ges.*) Gerade auf diesem Gebiet hat sich in starkem Maße die Praxis von der Duden-Regel entfernt. Wenn vor allem mehrfach zusammengesetzte Wörter abgekürzt werden und dies entsprechend der

Duden-Regel geschieht, würden sehr lange und wenig platzsparende Formen entstehen. Deshalb hat sich in dieser Gruppe eine Abkürzungsmethode herausgebildet, der Klarheit und Sinnfälligkeit nicht abgesprochen werden kann. Bei zusammengesetzten Wörtern werden bei der Abkürzung jeweils die Anfangsbuchstaben jedes entscheidenden Bestandteils mit Anfangsgroßschreibung und ohne Bindestrich und ohne Zwischenraum aneinandergefügt; die Anfangsgroßschreibung soll dabei den Beginn der einzelnen Bestandteile aus den zugrundeliegenden Auflösungen kennzeichnen; z.B.: *ORegMedR (=Oberregierunsmedizinalrat*; nach Duden: *O.-Reg.-Med.-R.*). Enden solche Abkürzungen mit Großbuchstaben, wird kein Schlußpunkt gesetzt; enden sie mit einem Kleinbuchstaben, wird ein Punkt gesetzt, z.B. *ArchG (= Architekten Gesetz)*, *BAnw. (= Bundesanwaltschaft)*. In verschiedenen Bereichen (Post, Bahn, Verwaltung) wird aber auch in diesem Falle auf den Schlußpunkt verzichtet.

Die obigen Ausführungen lassen erkennen, daß bei der Behandlung von Abkürzungen eine Vielzahl von Regeln und vor allem Ausnahmen berücksichtigt werden müssen. Ein weiteres Problem stellt die Tatsache dar, daß eine Abkürzung im allgemeinen für die Singular- und die Pluralform, sowie für die übrigen Deklinationsformen der Auflösung(en) steht. Die jeweils richtige Übersetzung ergibt sich dann aus dem (syntaktischen) Kontext.

Die erschöpfende Behandlung dieser Fülle würde den Rahmen dieser Studienarbeit sprengen. Das Folgende beschränkt sich deshalb auf die wesentlichen und gebräuchlichen Abkürzungen.

Die Abkürzungen werden jeweils mit ihrer entsprechenden ausgeschriebenen Form in Tabellen gespeichert. Für verschieden Arten von Abkürzungen sind

dabei unterschiedliche Tabellen vorgesehen. Die BenutzerIn kann zusätzliche Eintragungen direkt in der Datei `lib/german/german_token_to_words_lists.scm` vornehmen. Eine Aufstellung der Tabellen ist in Abschnitt 3.7 zu finden.

Ähnlich wie bei der Bearbeitung von Zahlen. läßt sich die Behandlung von Abkürzung in den Bereich der Erkennung im laufenden Text und den Bereich der Expansion in eine aussprechbare Form unterteilen. Im folgenden werden die behandelten Arten von Abkürzungen erklärt:

1. Token die der Duden-Regel entsprechen sind Abkürzungen
2. Maßeinheiten sind Abkürzungen
3. Token der Länge eins sind Abkürzungen (z.B. *S* für Süden).
4. Token die nur aus Konsonanten ohne y bestehen sind Abkürzungen (z.B. *vgl.* für *vergleiche*).
5. Token die an einer Stelle ungleich der ersten einen grossen Buchsatben besitzen können Abkürzungen sein (z.B. *USA*).
6. Token denen ein Punkt folgt können Abkürzungen sein.

3.3.1 Abkürzungen nach Duden-Regel

Abkürzungen nach Duden-Regel (wie *Ver.-Ges.*) werden in der Funktion `german_token_to_words` ganz am Anfang erkannt und von der Funktion `ger_lookup_comb_abbr` rekursiv aufgesplittet (in Vers. und Ges.) und dann einzeln in den entsprechenden Tabellen gesucht. Wird keine Übersetzung gefunden werden sie buchstabiert.

3.3.2 Maßeinheiten

Die Erkennung und Expansion von Maßeinheiten (wie Beispielsweise *kHz*) unterscheidet sich wesentlich von den restlichen Abkürzungen: Soll eine Abkürzung einer Maßeinheit als solche erkannt werden, muß das vorangehende Token eine Zahl gewesen sein und die Abkürzung muß einem der beiden rgulären Ausdrücke `ger_masseinheit_teststring` oder `ger_masseinheit_teststring2` entsprechen. Ist dies der Fall, wird sie aufgrund der Informationen in den Tabellen `ger_abbr_masseinheiten_dim_tab` und `ger_abbr_masseinheiten_tab` übersetzt. In der ersten Tabelle stehen die mögliche Dimensionen der Maßeinheiten (*pico, nano, kilo, tera...*), in der zweiten Tabelle die Maßeinheiten ohne die entsprechende Dimension. Das Vefahren mit den regulären Ausdrücken wurde wegen der möglichen Kombination aller Dimensionsangaben mit den verschiedenen Einheiten gewählt. Dadurch müssen nicht für jede neu eingetragene Maßeinheit auch alle Kombinationene eingetragen werden. Man muß lediglich die beide regulären Ausdrücke anpassen.

Die Erkennung von Abkürzungen der folgenden drei Arten wird in der Funktion `ger_check_for_abbr` durchgeführt, die im Falle, daß das Token eine Abkürzung ist, gleich die passende Expansion mit Hilfe der Funktion `ger_translate_abbr` bestimmt.

3.3.3 Abkürzungen der Länge eins

Token, die nur aus einem Buchstaben bestehen, sind immer Abkürzungen. Sie werden allerdings nicht expandiert, da sie meist sehr viele verschiedene

Bedeutungen haben.

3.3.4 Abkürzungen die nur aus Konsonanten bestehen

Token die nur aus Konsonanten bestehen sind in jedem Fall Abkürzungen, da sie im Deutschen nicht aussprechbar sind.[11] Einen Sonderfall ist hierbei der Buchstabe *y* (wie Beispielsweise in *Wyhl*). Er wird deshalb aus der Menge der Konsonanten herausgenommen.

Ist eine Abkürzung nach diesem Kriteium erkannt, wird sie in den Abkürzungstabellen nachgeschaut (dies erledigt die Funktion `ger_translate_abbr`. Wird keine passende Übersetzung gefunden, wird sie buchstabiert.

3.3.5 Abkürzungen aus Grossbuchstaben

Token, die an einer Stelle, ungleich der ersten, einen Großbuchstaben haben (z.B. *USA, NATO ...*), werden als Abkürzungen erkannt, falls sie in einer der Abkürzungstabellen gespeichert sind. Ist dies nicht der Fall werden sie wie normale Worte ausgesprochen. Dieses Vorgehen wurde gewählt, da in vielen Texten normale Wörter in großen Buchstaben geschrieben werden, um sie hervorzuheben. In Zeitungstexten wird oft die Herkunft eines Berichtes durch Voranstellen des Herkunftsortes in Großbuchstaben angezeigt (z.B. *SYDNEY (dpa) - der Leadsinger ...*).

Ist ein großgeschriebenes Wort in einer der Abkürzungstabellen vorhanden, wird es, gemäß der dort angegebenen Übersetzung, expandiert.

[11] besser wäre an dieser Stelle eine ausführlichere Überprüfung auf Aussprechbarkeit.

3.3.6 Abkürzungen mit nachfolgendem Punkt

Falls ein Token als `punc`-Merkmal einen Punkt hat, wird es in der Tabelle der Abkürzungen mit Punkt nachgeschaut. Ist es darin nicht enthalten, wird davon ausgegangen, daß der Punkt ein Satzende markiert und daß Token wird der Behandlung von regulären Worten zugeführt. Sonst wird es nach den Angaben in der Tabelle expandiert und der Punkt, um die spätere Phrasierung nicht zu beeinträchtigen, aus dem `punc`-Merkmal gelöscht.

3.3.7 Mehrdeutige Token

Ein besonderes Problem stellen Abkürzungen dar, welche auch gleichzeitig ein reguläres Wort sein könnten. Ein Beispiel hierfür ist *Art.*. Dies kann sowohl die Abkürzung für *Artikel* sein als auch das Wort *Art* am Satzende darstellen. Zur Erkennung dieser Art von Abkürzungen muß deren Umgebung untersucht werden. Möglich wäre dies z.B. mit Hilfe eines Entscheidungsbaumes, ähnlich dem zur Unterscheidung von Ordinalzahlen und Kardinalzahlen am Satzende (vgl. 3.2.10). Zu beachten ist, daß solche Abkürzungen nicht einfach in die Tabelle der Abkürzungen mit Punkt eingetragen werden dürfen, da dies zu falschen Ergebnissen führen kann.

3.4 Behandlung von Interpunktion und Wortzwischenräumen

Die Interpunktionszeichen werden bereits im Modul `Text` vom Wort abgetrennt und als Feature im Tokenstrom gespeichert. Auf sie kann, wie in Abschnitt 2.3.2 beschrieben, zugegriffen werden. Interpunktionszeichen werden

unter anderem dazu verwendet Satzpausen zu bestimmen (vgl. Abschnitt 2.3.6). In der Textvorverarbeitung dienen sie in erster Linie zur Bestimmung von Ordinalzahlen (vgl. 3.2.10) und Abkürzungen (vgl. 3.3).

Auch Wortzwischenräume werden wie in 2.3.2 beschrieben als Feature im Tokenstrom gespeichert.

3.5 Expansion von Sonderzeichen

Alle nicht-alphanumerischen Zeichen sind Sonderzeichen und werden als solche behandelt. Isoliert stehende Sondezeichen werden in der Regel ausgesprochen. Ausanahme hiervon ist der Bindestrich. Tritt ein Bindestrich isoloiert auf, d.h. von Wortzwischenräumen umgeben, wird er nicht Ausgesprochen, sondern an dessen Stelle eine kurze Pause eingefügt. Alle anderen isolierten Sonderzeichen werden gemäß der Tabelle `geraman_symbol_tab` übersetzt. Sonderzeichen, die bereits bei der Tokenisierung abgetrennt wurden, werden grundsätzlich nicht gesprochen. Enthält ein Token im inneren Sonderzeichen, so wird es an der Stelle, an dem das Sonderzeichen steht, in zwei Teile zerlegt, die durch rekursiven aufruf von `german_token_to_words` weiter bearbeitet werden. Das Sonderzeichen wird dann, je nach dem um welches Sonderzeichen es sich handelt, entweder ausgesprochen oder auch nicht. Ausgesprochen werden z.B. Ausrufezeichen, Klammeraffe usw.; nicht ausgesprochen werden dagegen das Miuszeichen, Anführungszeichen, sowie diverse Klammern.

3.6 Verwendete Hilfsfunktionen

Zur Erkennung und Umsetzung von Ziffernfolgen, Abkürzungen und andere Besonderheiten im Text, werden verschiedene Hilfsfunktionen verwendet.

Diese Funktionen sind, bis auf `german_parse_cardinal`, in Scheme implementiert. Ihre Definition findet sich in `lib/german/grman_token_to_words_tools`. `german_parse_cardinal` ist in C++ in der Datei `src/modules/german_text/german_text.cc` implemetiert. Im folgenden werden die einzelnen Funktionen kurz beschrieben.

`(german_parse_cardinal NUMBERSTRING)`: Übersetzt den übergebenen Ziffernstring NUMBERSTRING in eine Liste von Worten. NUMBERSTRING darf nicht länger als 24 Zeichen sein und nur Ziffern enthalten. Führende Nullen werden nicht übersetzt. Ist NUMBERSTRING länger als 24 Zeichen wird `(unknowm)` zurückgegeben. Enthält die Ziffernfolge etwas anderes als Ziffern, ist das Ergebnis undefiniert. Die zurückgegebene Wortliste repräsentiert die ausgeschriebene Form der Kardinalzahl, die durch NUMBERSTRING dargestellt ist.

`(german_parse_1char NAME)`: Übesetzt ein einzelnes Zeichen NAME in eine aussprechbare Form. Ziffern werden in das entsprechende Wort übersetzt, Buchstaben bleiben erhalten und Sonderzeichen werden gemäß der Tabelle `ger_char_tab` übersetzt. Zeichen, die weder Buchstaben, Ziffern noch in der Tabelle enthalten sind, werden mit `(unknown)` übersetzt.

`(german_parse_charlist NAME)`: Ist die rekursive Erweiterung von `german_parse_1char`. Die Zeichekette NAME wird zeichenweise aufgespalten und mit `german_parse_1char` übersetzt. Diese Funktion wird zum Buchstabieren verwendet. Zurückgegeben wird eine Liste von Worten und Buchstaben.

`(splice_string NAME SEP)`: Diese Funktion entfernt alle Vorkommen des Zeichens SEP im String NAME. Sie wird z.B. dafür verwendet um bei Kardinalzahlen, die mit Punkten gruppiert sind, die Punkte zu entfernen.

`(define (split_string NAME SEP)`: wird nicht mehr verwendet.

`(sep_string NAME SEP SEPWORD)`: wird nicht mehr verwendet.

`(define (german_parse_ordinal NUMBERSTRING SUFFIX))`: Übersetzt den übergebenen Ziffernstring NUMBERSTRING in eine Liste von Worten. NUMBERSTRING darf nicht länger als 24 Zeichen sein und nur Ziffern enthalten. Führende Nullen werden nicht übersetzt. Ist NUMBERSTRING länger als 24 Zeichen, wird `(unknowm)` zurückgegeben. Enthält die Ziffernfolge etwas anderes als Ziffern, ist das Ergebnis undefiniert. Die zurückgegebene Wortliste repräsentiert die ausgeschriebene Form der Ordinalzahl die durch NUMBERSTRING dargestellt ist. Die Endung der Ordinalzahl wird durch den String SUFFIX angegeben. Intern wird die Funktion `german_parse_cardinal` verwendet.

`(german_parse_fractal NUMBERSTRING)`: Übersetzt den übergebenen Ziffernstring NUMBERSTRING in eine Liste von Worten. NUMBERSTRING darf nicht länger als 24 Zeichen sein und nur Ziffern enthalten. Führende Nullen werden nicht übersetzt. Ist NUMBERSTRING länger als 24 Zeichen wird `(unknowm)` zurückgegeben. Enthält die Ziffernfolge etwas anderes als Ziffern, ist das Ergebnis undefiniert. Die zurückgegebene Wortliste repräsentiert die ausgeschriebene Form der Teilungszahl die durch NUMBERSTRING dargestellt ist. Intern wird die Funktion `german_parse_cardinal` verwendet.

`(german_fetch_currency UTT TOKEN NAME)`: Bestimmt die Währungseinheit der momentan behandelten Zahl, indem das folgende Token mit Hilfe der `ger_abbr_currency_tab` übersetzt wird.

`(ger_tok_roman_to_numstring ROMAN)`: Übersetzt eine römische Zahl bis 50 in eine arabische Zahl.

`(ger_find_month_from_number UTT TOKEN STRING-NUMBER)`: Übersetzt die Monatsnummer in den entstprechenden Monat. Wird zur Zeit nicht verwendet.

`(ger_abbr_lookup ABBR TAB)`: Diese Funktion schaut die Abkürzung ABBR in der Abkürzungstabelle TAB nach und gibt das Ergebnis zurück. Bis jetzt werden Abkürzungen noch in sogenannten *Assoziationslisten* gespeichert. Wenn die Ab-

kürzungstabellen sehr groß werden, sollten statt dessen *Hash-Tabellen* verwendet werden. Dann muß diese Funktion entsprechend angepaßt werden.

`(ger_abbr_getkeys TAB)`: Diese Funktion gibt die Liste aller Schlüssel zurück, die in der Tabelle TAB gespeichert sind.

`(ger_abbr_getvals TAB)`: Diese Funktion gibt die Liste aller Werte zurück, die in der Tabelle TAB gespeichert sind.

`(ger_lookup_comb_abbr NAME)`: Diese Funktion dient zur Expansion von kombinierten Abkürzungen nach der Duden-Regel.

`(ger_check_for_abbr UTT TOKEN NAME)`: Diese Funktion überprüft, ob NAME eine Abkürzung ist. Wenn ja, wird die dazugehörige Übersetzung gleich mit zurückgegeben..

`(ger_translate_abbr UTT TOKEN NAME)`: Wird von `ger_check_for_abbr`: benutzt, um Abkürzunen in ihre volle Form zu übersetzen.

3.7 Verwendete Tabellen und Listen

: Insbesonder zur Übersetzung von Abkürzungen werden eine Menge von Tabellen und Listen verwendet. Sie sind in der Datei `lib/german/grman-_token_to_words_lists` definiert und sollten von der BenutzerIn angepaßt werden.

Es Folgt eine Aufstellung mit kurzer Beschreibung:

`ger_abbr_currency_tab`: In dieser Tabelle sind Abkürzungen von verschieden Währungen und deren Übersetzung gespeichert.

`ger_abbr_without_point_tab`: In dieser Tabelle sind Abkürzungen und der Expansionen gespeichert, welche nicht durch einen Punkt beendet werden.

ger_abbr_with_point_tab: In dieser Tabelle sind Abkürzungen und der Expansionen gespeichert, welche durch einen Punkt beendet werden.

ger_abbr_at_eos_tab: In dieser Tabelle sind Abkürzungen und deren Übersetzung gespeichert, welche durch einen Punkt beendet werden und fast immer am Ende eine Satzes auftreten.

ger_abbr_masseinheiten_tab: In dieser Tabelle sind Abkürzungen von Maßeinheiten (ohne Dimension) und deren Übersetzung gespeichert. Durch Kombination mit der entsprechenden Dimension aus ger_abbr_masseinheiten_dim_tab ergeben sich dann alle möglichen Maßeinheiten (vgl. 3.3.2).

ger_abbr_masseinheiten_dim_tab: In diese Tabelle werden die Dimensionen der Maßeinheiten eingetragen. Zusammen mit ger_abbr_masseinheiten_tab ergeben sich dann alle mögliche Maßeinheiten.

ger_masseinheit_teststring: ger_masseinheit_teststring2 Diese beiden regulären Ausdrücke ermöglichen die Überprüfung, ob ein Token eine Maßeinheit ist oder nicht, ohne alle möglichen Kombinationen von Maßeinheiten und Dimensionen in einer Tabelle zu speichern.

ger_abbr_months_tab: In dieser Tabelle werden Abkürzungen von Monatsname mit ihren Vollformen assoziiert.

ger_abbr_days_tab: Das selbe wie ger_abbr_months_tab nur für Wochentage

ger_symbols_tab: Hier sind alle nicht-alphanumerischen Zeichen mit entsprechender Übersetzung gespeichert. Diese Tabelle wird z.B. zum von der Funktion german_parse1_1char benutzt

ger_digits_tab: Die Ziffer 0-9 und ihre Expansion.

ger_char_tab: Wird aus den beiden Tabellen ger_digits_tab ger_symbols_tab generiert,

ger_abbr_compl_tab: Wird aus ger_abbr_days_tab ger_abbr_without_point_tab ger_abbr_months_tab ger_abbr_masseinheiten_dim_tab ger_abbr_masseinheiten_tab ger_abbr_with_point_tab und ger_abbr_currency_tab generiert

german_satzanfang: Ist eine Mange von Worten, die, sofern sie richtig geschrieben sind, nur am Anfang eines Satzes stehen können. Die Menge wird dazu verwendet einen rudimentäre Satrgrenzenbestimmung durchzuführen (vgl. 3.2.10).

german_king_title: Eine Liste von Worten, die einen Titel, ähnlich dem des Königs, darstellen. Sie werden dazu verwendet Äußerungen, wie *König Rio II*, korrekt expandieren zu können.

german_queen_title: Wie german_king_title, aber für Königinnen.

german_king_name: Liste von Namen wichtiger Könige, Kaiser. Päpste ...

german_queen_name: Liste von Namen wichtiger Königinnen, Kaiserinnen, Päpstinnen ...

german_year_indicator: Worte die vermuten lassen, daß die nachfolgende Zahl eine Jahreszahl ist.

german_ordinal_prediction_tree: Dieser Entscheidungsbaum ist von Hand erstellt und dient dazu, die Endungen von Ordinalzahlen aus dem Kontext der umgebenden Token zu bestimmen. Werden die Äußerungen in späteren Vesrionen syntaktisch analysiert, können die Endungen aus dem syntaktischen Kontext sicherer bestimmt werden.

4 Einbinden der deutschen Morphologie in Festival

Um Komposita, welche nicht im Aussprache-Lexikon gespeichert sind, in Lautsprache zu übersetzen, sollen sie zunächst in ihre Stämme zerlegt werden. Danach können diese Stämme getrennt im Lexikon nachgeschlagen werden, um sie danach wiederum phonolgisch zu konkatenieren.

Dazu soll die deutsche Morphologie DMOR, die am IMS u.a. von Anne Schiller und Andreas Eisele, entwickelt und mit XEROX-Tools[12] zu einem endlichen Automat (finite state transducer, FST) übersetzt wurde, in Festival eingebunden werden.

Um DMOR in Festival zu benutzen, wird ein Teil der von Andreas Eisele implementierten FST-API (siehe Datei `src/modules/dmor/fst.h`) als Schemefunktion zur Verfügung gestellt. Dafür sind drei Schritte notwendig (vgl. [1] Kapitel *Programming*):

1. das Unterverzeichnis `dmor` unter `src/modules/` muß erstellt und in der Datei `src/modules/OTHER_DIRS` bekannt gemacht werden. Das Makefile in `src/modules/` sucht in dieser Datei nach weiteren Modulen, die bei der Compilierung von Festival dazugelinkt werden müssen

2. die Datei `dmor.C` im Verzeichnis `src/modules/dmor` wird erstellt. Sie enthält außer den Funktionen, welche die Schnittstelle zwischen der FST-API und Festival darstellen, die Funktion `festival_dmor_init`.

[12] Durch einen entsprechenden Lizenzvertrag zwischen XEROX und dem IMS, steht alle, mit Hilfe der XEROX-Tools entwickelte Software, unter derr Lizenz von XEROX. Somit auch die deutsche Morphologie DMOR. Sie darf deshalb nur hausintern verwendet werden.

Diese wird benötigt um die C-Funktionen auf Scheme-Ebene aufrufen zu können.

3. im Unterverzeichnis `src/modules/dmor/` muß ein Makefile erstellt werden. Der Inhalt kann weitgehend aus dem Makefiles eines anderen Moduls (z.B. `src/modules/Text/Makefile`) übernommen werden.

Aus der FST-API sind für die Festival-spezifische Benutzung von DMOR vor allem zwei Funktionen interessant:

1. `load_dmor_w1()`: Mit dieser Funktion wird die Automatentabelle von DMOR sowie die zugehörige Datei mit den Gewichten in den Speicher geladen.

2. `weighted_fst_requets`: Mit dieser Funktion wird die Automatentabelle auf der Eingabe simmuliert und das Ergebnis zurückgegeben.

Mit Hilfe dieser Funktionen wird die Schemefunktion `dmor.infl` implemetiert. Mit ihr können Worte morphologisch analysiert oder aus Morphemen Wort generiert werden. Wie dies genau funktioniert ist in [3] und [4] beschrieben. Hier nur ein kurzer Beispielsdialog:

```
festival> (dmor.infl '("Donaudampfschiffahrt"))
("0.250000 Donau=Dampf=Schiff=Fahrt+NN.Fem.Nom.Sg
0.250000 Donau=Dampf=Schiff=Fahrt+NN.Fem.Gen.Sg
0.250000 Donau=Dampf=Schiff=Fahrt+NN.Fem.Dat.Sg
0.250000 Donau=Dampf=Schiff=Fahrt+NN.Fem.Akk.Sg
")
festival> (dmor.infl '("Donau=Dampf=Schiff=Fahrt+NN.Fem.Gen.Pl" "g"))
("1.000000 Donaudampfschiffahrten
")
festival>
```

Das "g" im zweiten Aufruf von `dmor-infl` zeigt an, daß ein Wort generiert werden soll.

5 Ausblick

Die deutsche Version des Festival Sprach Synthese Systems steht noch ganz am Anfang ihrer Entwicklung. Bis jetzt wurden hauptsächlich vorhandene Werkzeuge, wie z.B. der Diphon-Synthesizer aus dem MBROLA Projekt (vgl. [24] [17] [18]) eingesetzt. Dadurch wurde eine Stufe erreicht, in der eine komplette, wenn auch in der Qualität nicht immer überzeugende, TTS-Synthese möglich ist. In Zukunft können nun neue Ideen eingebaut, getestet und optimiert werden. Dadurch ist es möglich, sowohl einzelne Verfahren zur Sprachsynthese wie auch das Gesamtsystem weiterzuentwickeln. Im folgenden werden einige Aufgaben beschrieben, welche in nächster Zukunft in Angriff genommen werde sollten.

5.1 Textvorverarbeitung

Bei der Textvorverarbeitung bleiben noch einige Probleme offen. Insbesondere die von Hand erstellten Entscheidungsbäume zur Disambiguierung von Zahlen und zur Bestimmung der Endung von Ordnungszahlen sind Heuristiken und arbeiten nicht immer einwandfrei. In Hinblick auf eine baldige Einbindung der stochastischen Grammatik (vgl. Abschnitt 5.2.4) ist es sinnvoll verschiedene Aufgaben der Textvorverarbeitung, welche auch für die Grammatik erledigt werden müssen, aufeinander abzustimmen. Andere Probleme, die sich speziell aus der TTS-Synthese ergeben, können oft erst nach einer syntaktischen Analyse sinnvoll behandelt werden. So ist z.B. die Umsetzung von Ordnungszahlen, die in Kasus, Numerus und Genus Flektiert werden nüssen, nach erfolgter syntaktischer Analyse, einfach zu bewerkstelligen. Es bietet sich also an, die Textvorverarbeitung in zwei Teile aufzuspalten:

1. In den Teil, welcher auch zur syntaktischen Analyse notwendig ist. Dazu gehören unter anderem die Tokenisierung und die Bestimmung von Satzgrenzen[13]. Dieser Teil muß auf die syntaktische Analyse abgestimmt sein.

2. In den Teil, der speziell für die TTS-Synthese benötigt wird, also die Umsetzung von Ziffernfolgen, Abkürzungen und Sonderzeichen in eine aussprechbare Form. Hierzu liefert die syntaktische Analyse wichtige Informationen, welche zur Disambiguierung eingesetzt werden können.

Bevor also weiter Arbeit in die Verbesserung der o.g. Heuristiken gesteckt wird, sollte man überlegen, ob diese nicht vieleicht bald überflüssig werden.

5.2 Morphosyntaktische Analyse

Mit Hilfe einer Morphosyntaktische Analyse könnte das System wesentlich verbessert werden. Wie diese aussehen könnte wir im folgenden grob skizziert:

5.2.1 Morphologische Analyse versus POS-Tagging

Zur Bestimmung von Aussprache, Satzpausen und Prosodie ist es wichtig, die Worte in grammatikalische Kategorien einzuordnen. Dafür gibt es im wesentlichen zwei Möglichkeiten:

1. Eine "echte" morphologische Analyse, welche aufgrund eines Stamm- und Endungslexikons, sowie diverser morphologischer Regeln, ein Wort

[13]Eine mögliches Verfahren zur Bestimmung von Satzgrenzen ist in [13] beschrieben

zerlegt und Wortart, sowie dessen syntaktische Funktion (Genus, Kasus ...) bestimmt. Eine solche morphologische Analyse ist meist mehrdeutig und muß mittels einer syntaktische Analyse disambiguirt werden. Der Vorteil bei der morphologischen Analyse ist, daß sie die, für kompositions- und derivationsreichen Sprachen wie dem Deutschen, gleich eine Zerlegung des Wortes miliefert, und so die Umsetzung in eine phonetische Transkription unterstützt (siehe oben).

2. Ein Part of speech Tagger betimmt die Wortart dagegen mit Hilfe von statistischen Modellen (vgl. [14]). Er liefert zwar eine eindeutige Kategorisierung, hilft aber bei der Zerlegung von Komposita nicht weiter.

Da geplant ist die stochastische Syntax aus dem *SPARKLE* Projekt (vgl. [26]) einzusetzen und diese auf DMOR aufbaut ist der erstgenannte Weg vorgesehen.

5.2.2 Morphologische Analyse

Die morphologische Analyse hat im wesentlichen zwei Aufgaben: Erstens dient sie der Bestimmung der Wortarten und bietet damit die Grundlage für eine syntaktische Analyse. Zweitens liefert sie eine morphologische Zerlegung der Worte und bildet dadurch eine Ausgangsbasis für eine Morphem-basierte Umsetzung in eine phonetische Transkription (siehe nächster Abschnitt).

Angesichts der Tatsache, daß die zur Einbindung vorgesehenen stochastische Syntax (vgl. 5.2.4) ebenfalls die deutsche Morphologie *DMOR* zur Vorverarbeitung verwendet, muß überlegt werden, wie diese Komponenten aufeinander abgestimmt werden können. Da für die syntaktische Analyse sowieso jedes Wort morphologisch analysiert werden muß, könnte es sinnvoll sein,

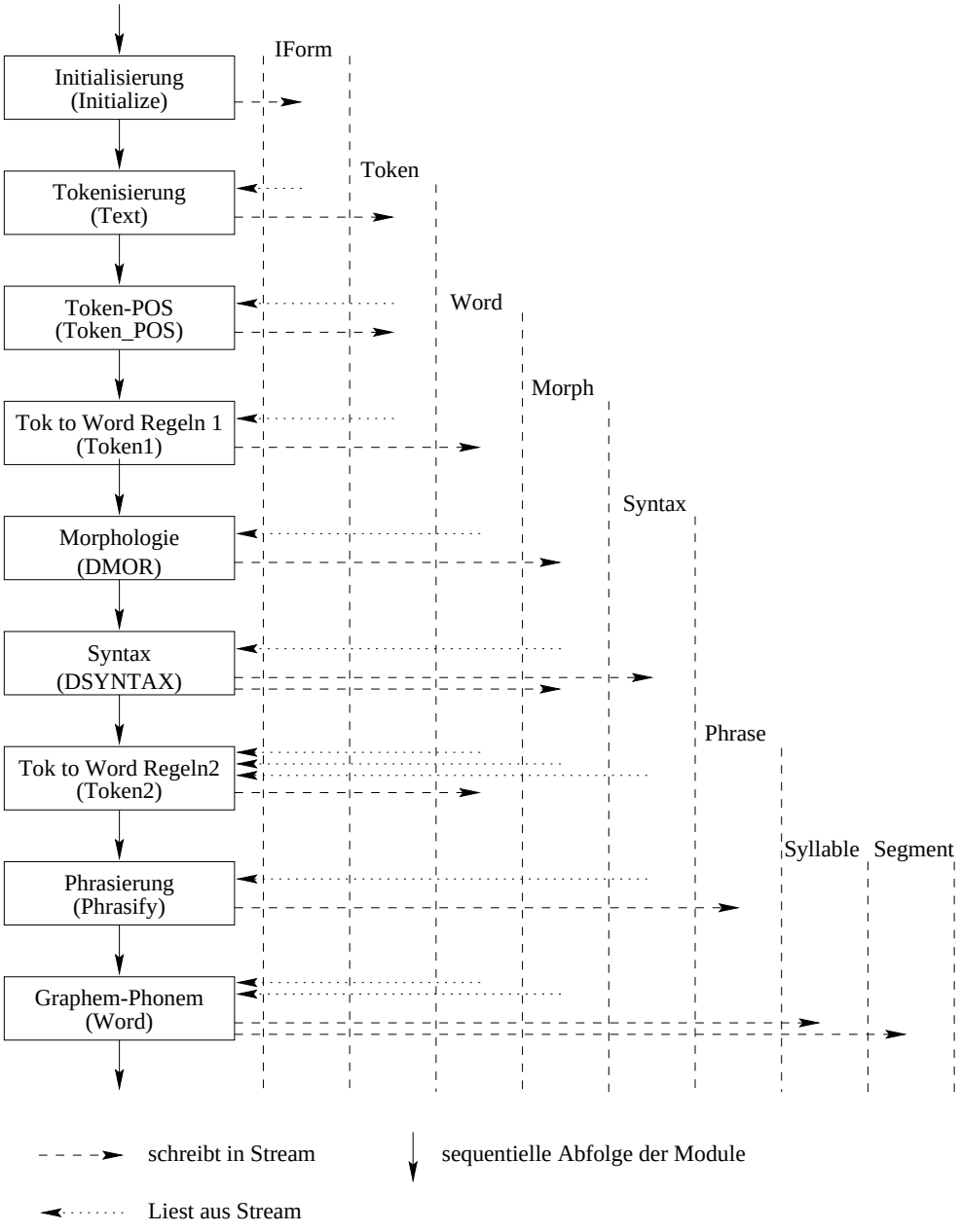

Abbildung 4: Die Module des Festival TTS-Systems und deren Zugriffe auf die Utterance-Struktur (unvollständig). Mit den morphosyntaktischen Modulen

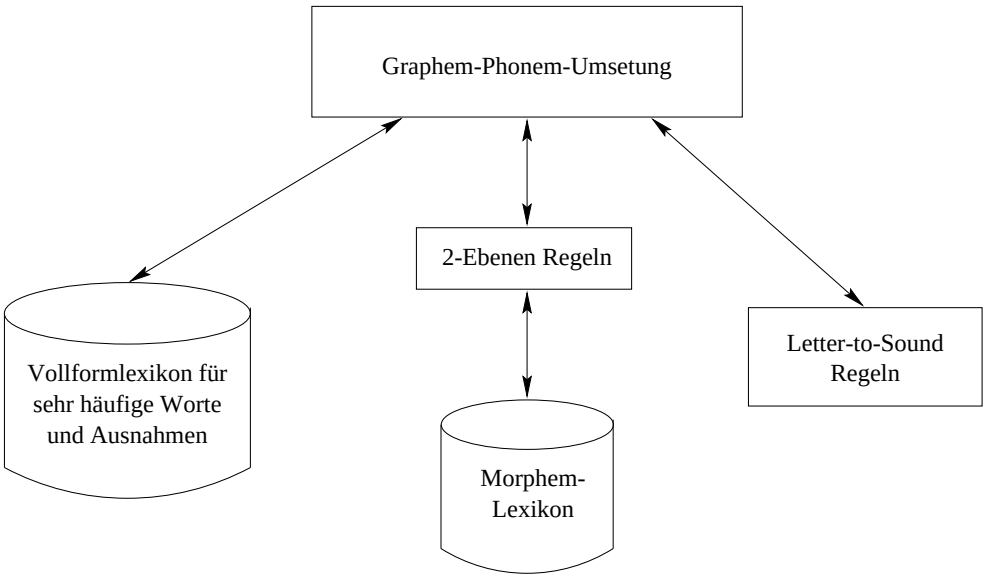

Abbildung 5: Das Graphem-Phonem Modul, wie es bald in Festival vorhanden sein könnte.

die Morphologie als eigenes Modul in Festival einzubinden und einen zusätzlichen Strom von morphologisch analysierten Wörtern in die Utterance aufzunehmen. Die Syntax kann dann wiederum auf diesen Strom zugreifen und eventuelle Mehrdeutigkeiten beseitigen. Diese Vorgehensweise ist in Bild 4 dargestellt.

Die Graphem-Phonem Umsetzung wird dann nicht mehr in erster Linie auf ein Vollformlexikon aufbauen (vgl. 2.3.7). Wie Abbildung 5 ersichtlich, sind dann nur noch die wichtigsten Worte und Ausnahmem im Vollformlexikon enthalte. Wird dort ein Wort nicht gefunden, wird es morphologisch nachgeschlagen. Erst wenn das fehl schlägt kommen die Letter-to-Sound Regeln zum Einsatz.

5.2.3 Ausspracheregeln mit 2-Ebenen-Modell

Im Bezug auf die Graphem-Phonem Konversion und der erfolgten Zerlegung der Worte mit DMOR wäre daran zu denken, eine solchen endlichen Automat, der nach dem 2-Ebenen-Modell (vgl. [4]) aufgebaut ist, nicht nur für die morphologisch Analyse, sondern auch für die anschlißende Generierung der Phonetischen Repräsentation einzusetzen. Diese Vorgehensweise ist beispielsweise in [15] Abschnitt 3.3.7 beschrieben.

5.2.4 Syntaktische Analyse

Die syntaktische Analyse hat im wesentlichen zwei Aufgaben: Sie hilft einerseits, die Mehrdeutigkeiten der morphologischen Analyse zu beseitigen, auf der anderen Seite liefert sie wichtige Informationen zur Prosodiesteuerung und ist Basis für eventuelle semantische oder pragmatische Analysen (vgl. Abbildung 2).

Vorgesehen ist hier die Einbindung des probabilistischen Parsers aus dem SPARKLE Projekt (vgl. [26]). Ansprechpartner am IMS ist Prof. Mats Rooth. Wie die Syntax in das Gesamtsystem eingebettet werden könnte zeigt Abbildung 4.

5.3 Synthese

Für die Synthese ist der Einsatz eines Moduls vorgesehen, welches nach dem *unit selection* Verfahren arbeitet (vgl. [16] und [23]).

Literatur

[1] Black, Alan W. und Paul Taylor *The Festival Speech Synthesis System*. System documentation, edition 1.2 for Festival Version 1.2.0. Stand: 5. September 1997.

[2] Taylor, Paul *Edinburgh Speech Tools Library*. System documentation, edition 1.0 for Speech Tools Library Version 1.0.0 Stand: 24. Januar 1997.

[3] Schiller, Anne *DMOR: Benutzeranleitung*. Interner Report. Institut für maschinelle Sprachverarbeitung, Universität Stuttgart, 1995.

[4] Schiller, Anne *DMOR: Entwicklerhandbuch*. Interner Report. Institut für maschinelle Sprachverarbeitung, Universität Stuttgart, 1995.

[5] Kommenda, Markus *Automatische Wortstrukturanalyse für die akustische Ausgabe von deutschem Text*, Dissertation. Technische Universität Wien. Mai 1991.

[6] Bauer, Silvia *Textvorverarbeitung für ein Sprachsynthesesystem*, Diplomarbeit. Technische Universität Wien. April 1987.

[7] Wall, Larry und Randal L. Schwartz *Programming Perl*, O'Reilly & Associates, Inc., Sebastopol, März 1992.

[8] Norvig, Peter *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp*. San Francisco, California: Morgan Kaufmann, 1992 .

[9] Pfister, B., C. Traber, *Text-to.Speech Synthesis: An Introduction and A Case Study,* in: Eric Keller (Hsg.), *Fundamentals of Speech Synthesis*

and Speech Recognition: Basic Concepts, State of the Art and Future Challenges, Wiley, Cichester, 1994.

[10] Allen, Jonathan *Overview of Text-to-Speech Systems* in *Advances in speech synthese process*, 1992.

[11] Schwarz, Diemo *Aufbau von Text-to-Speech Systemen*, Ausarbeitung zum Seminar *Automatische Sprachsynthese*, WS 96/97

[12] Werlin, Josef *Duden Wörterbuch der Abkürzungen*, Bibliographisches Institut Mannheim[u.a.], 1971.

[13] Palmer, David D. u. Marti A. Hearst *Adaptive Multilingual Sentence Boundary Disamiguation* in *Computational Linguistics*, published quarterky by the MIT-press, Volume 23, Number 2, Juni 1997.

[14] DeRose, S. *Grammatical category disambiguation by statistical optimization* in *Computational Linguistics*, 14:31-39, 1988.

[15] Traber, Chritof *SVOX: The Implementation of a Text-to-Speech System fo German*, Institut für Technische Informatik und Kommunikationsnetze, ETH-Zürich, Zürich: vdf, 1995.

[16] Hunt, K. und Black, A. *Unit selection in a concatenative speech synthese system using a large database*, ICASSP-96, vol. 1 pp 373-376, Atlanta, Georgia 1996.

[17] Dutoit, T.; V. Pagel, V. Pierret, N. Ierret, F. Bataille und O. van der Vrecken *The MBROLA Project: Towards a Set of High-Quality Speech Synthesizers Free of Use for Non-Commercial Purposes*, Proc. ICSLP'96, Philadelphia, vol. 3, pp. 1393-1396.

[18] Dutoit, T. *An Introduction to Text-To-Speech Synthesis*, Kluwer Academic Publishers, Dordrecht Hardbound, ISBN 0-7923-4498-7 April 1997, 312 pp.

[19] Baayen, R. H., R. Piepenbrock und L. Gulikers *The CELEX Lexical Database (CD-ROM).* (die CD-ROM ist am IMS vorhanden) Linguistic Data Consortium, University of Pennsylvania, Philadelphia, PA, 1995.

Quellen im WWW: Aufgrund der raschen Entwicklung des Internets sind diese Quellen unter Umständen bereits nach kurzer Zeit nicht mehr Aktuell.

[20] Breitenbücher, Mark *Informatik und Umweltschutz*, Vortrag in Rahmen des Hauptseminars *Informatik und Gesellschaft* im Sommersemester 1997 am Institut für Informatik der Universität Stuttgart.
`http://www-stud.ims.uni-stuttgart.de/ breitenb/hauptsem.html`
Stand November 1997

[21] Black, Allan *The Festival Speech Synthese System.*
`http://www.cstr.ed.ac.uk/projects/festival.html`
Stand: November 1997

[22] Carrette, George J. *SIOD: Scheme in One Defun.*
`http://people.delphi.com/gjc/siod.html`
Stand: November 1997

[23] Hunt, K. und Black, A. *Unit selection in a concatenative speech synthese system using a large database.*
`http://www.cstr.ed.ac.uk/projects/festival/publications/1996/Black_1996_a.s.ps`
Stand: November 1997

[24] *The MBROLA PROJECT HOMEPAGE.*

`http://tcts.fpms.ac.be/synthesis/`

Stand: November 1997

[25] *Celex readme.*

`http://www.ldc.upenn.edu/readme_files/celex.readme.html`

Stand: November 1997

[26] *SPARKLE.*

`http://www.ims.uni-stuttgart.de/`

unter Forschungsinhalte? → Forschungsprojekte → SPARKLE

Stand: November 1997