# Building Chunk Level Representations For Spontaneous Speech in Unrestricted Domains: The CHUNKY System and Its Application To Reranking NBest Lists of a Speech Recognizer

M.S. Project Report

Klaus Zechner
Computational Linguistics Program
Department of Philosophy
Carnegie Mellon University
zechner@andrew.cmu.edu

Committee:
Alex Waibel *(LTI/SCS)*
Alon Lavie *(LTI/SCS)*
Nancy Green *(Robotics/SCS)*

May 5th, 1997

*To*
*Christine,*
*Betsy,*
*and*
*Matthew*

**Acknowledgements**

I am grateful for the help and suggestions from my three project advisors, for critical remarks from my classmates, and to Klaus Ries, Marsal Gavaldà, Torsten Zeppenfeld, Michael Finke, Petra Geutner, and Roni Rosenfeld for their support with data, tools, hints, and general help on my way. Specifically I want to thank those of my fellow classmates who were giving some of their valuable time in acting as "subjects" for my experiment.

I also want to thank in particular my parents, my family, and friends at home in Austria, and my friends here in Pittsburgh for their ongoing emotional and moral support during my years as a Masters student here at CMU.

This work was funded in part by grants of the Austrian Ministry for Science and Research (BMWF), the Verbmobil project of the Federal Republic of Germany, ATR – Interpreting Telecommunications Research Laboratories of Japan, and the US Department of Defense.

# Contents

# List of Figures

# List of Tables

**Abstract**

Parsing spontaneous speech has so far mainly been limited to narrow domain applications (e.g., scheduling of meetings, travel planning). In this work, a chunk based parsing approach is used for building a fast, robust, and shallow parsing system for spontaneous, conversational speech in unrestricted domains. The chunk parses produced by this parsing system can be usefully applied to the task of reranking Nbest lists from a speech recognizer, using a combination of chunk-based ngram model scores and chunk coverage scores.

The input for the system is Nbest lists generated from speech recognizer lattices. The hypotheses from the Nbest lists are tagged for part of speech, "cleaned up" by a preprocessing pipe, parsed by the Phoenix part of speech based chunk parser, and finally rescored using a backpropagation neural net trained on the chunk based scores. Finally, the reranked Nbest lists are generated.

The results are promising, in that the best performance on a randomly selected test set is a decrease in word error rate of 0.3 percent, mesaured on the new first hypotheses in the reranked Nbest lists. Although this gain is quite small, one has to take into account that more linguistic information could be used for further enhancements (e.g., a combination of subcategorization frames and selectional restrictions of verbal heads).

# Chapter 1

# Introduction

## 1.1 Problem Statement

In the area of parsing spontaneous speech, most work so far has primarily focused on dealing with texts within a narrow, well-defined domain. The main reasons behind this restriction have been to avoid having to maintain very large and complex grammars on the one hand, and large semantic knowledge sources on the other hand.

In my project I use a chunk parsing technique to develop a system which is able (i) to generate shallow syntactic structures from speech recognizer output, and (ii) to employ these representations for the task of reranking Nbest lists.

## 1.2 Significance to Computational Linguistics

There have been developments recently which encourage the investigation of the possibility of parsing speech in unrestricted domains. It was demonstrated that parsing natural language[1] can be handled by very simple, even finite-state approaches if one adheres to the principle of "chunking" the input into small and hence easily manageable constituents (see e.g. (Abney, 1996b; Light, 1996), a more extensive discussion follows in Chapter 2).

The goal of this project was to apply and sensibly adapt these ideas to spontaneous speech dialogues, where the input can be either speech recognizer hypotheses (e.g., Nbest lists, generated from the word lattice), or transcribed speech data. The development was done using the publicly available Switchboard corpus (Godfrey et al., 1992) and development/test data from recent Switchboard evaluations (Switchboard and Callhome databases).

The main challenge of this project was to show that even with a shallow analysis of the input (chunk parser) it is feasible to produce useful syntactic (constituent or chunk based) representations which can be used for generating

[1] mostly of the written, but also of the spoken type

scores to improve the word-accuracy of the speech recognizer by reranking its Nbest-list.

Additionally, the system runs quickly and is robust to human disfluencies and ungrammaticalities as well as shortcomings of various system components, such as imperfection in speech recognition, part of speech tagging and utterance segmentation.

While the focus of this project has been the application to Nbest-list reranking, representations generated by the chunk parser are potentially useful for many other areas in NLP, such as (i) information extraction from spontaneous speech, (ii) condensation or summarization of conversational speech, (iii) topic tracing for human-to-human dialogues, news broadcastings, and other spoken language sources, (iv) improved language-modeling through the availability of higher level dependencies, and (v) shallow machine translation for unrestricted domains.[2]

## 1.3 Motivation for Nbest List Reranking

There are several reasons, why reranking of the Nbest lists would be a good thing to do:

1. **Reducing the WER of the Speech Recognizer**: When one looks at some Nbest list, one has the intuitive feeling that some of the top (or high) ranked hypotheses "just don't make sense", in both syntactic and semantic terms, whereas further down in the list, more "meaningful" hypotheses would appear. (An example of that is illustrated in detail in Appendix B.2, where we demonstrate a sample run through the system using an excerpt from an Nbest list.) This intuitive observation gives thus rise for the hope that "the right kind" of higher level linguistic representation (be it symbolically or statistically derived, or be it a combination of both) can help to let these "more meaningful" hypotheses from further down in the list "move up" to some significant extent, s.t. *on average*, on a test set, the first hypotheses would have a lower WER than before the reranking process.

2. **Lattice Parsing**: For lattice parsing, basically the same argument applies: if we could make, e.g., the set of the top 20 hypotheses more "meaningful", this would result in a higher parsing accuracy, or, if our goal is more to extract semantic frames and/or information from the input rather than getting the "best match to the original reference", a higher semantic (or: frame) accuracy might be achievable.

3. **Database Queries**: In the case of restricted machine-computer-interaction tasks, like database queries, the reranking of more meaningful candidate

[2]These potentials will be particularly relevant once the next stage, the verb-argument subcategorization mapper is being implemented. But this was beyond the scope of this current project.

hypotheses makes even more sense than in the two applications just mentioned: The only purpose here is to understand relevant portions of the user's request, and not to exactly know the *wording* of his query. So we can even further depart from a notion like WER here, since it makes more sense to get *something* meaningful out from the query parser — even if it is quite off track — rather than no parse at all. Interactive error correction methods (Suhm et al., 1996) can then be applied to recover from these mis-understanding problems.

## 1.4 Difficulties in Nbest List Reranking

While we saw several reasons why reranking would be a sensible idea to go about, we also have to face a number of quite serious difficulties in doing this task.

1. A lower WER does not always correspond to better syntactic or semantic "well-formedness": While for some utterances the intuition about ill-formed high ranked hypotheses certainly is true, for other utterances this simple view does not hold. We see many cases in the data, where humans either don't agree in the relative "wellformedness" of the first and the true best hypothesis of an utterance, where they could make no judgement at all, or where their judgement would be in favor of the first best hypothesis with the *higher* WER.[3]

2. "Erratic" WER behaviour of individual utterances in the Nbest list: If one looks at the hypothesis WER as a function of its position in the Nbest list, one sees a picture that pretty much looks like random noise, see, e.g., Figure 1.1. This means that in effect there are no "regions" in the Nbest list which exhibit higher or lower WER but that the WER fluctuates in a strong way and that there are many "distractors" (i.e. hypotheses with a higher WER) close by "winning" hypotheses (with a lower WER). Picking the winning hypothesis and *not* run into a distractor is, therefore, a non-trivial task.

3. With the size of N in the Nbest list increasing, the "expected WER gain" (i.e. the average expected change in WER, if one picks any hypothesis from the list at random to swap it with first ranked one) drops; in our Devtest set, the expected WER gain is approx. -5%. This means in effect that any method we are using for reranking has first to bridge this gap between the expected WER gain and the zero baseline, before it can produce any improvements at all.

[3] Details about a human study explicitely devoted to this question will be given in section 4.6.

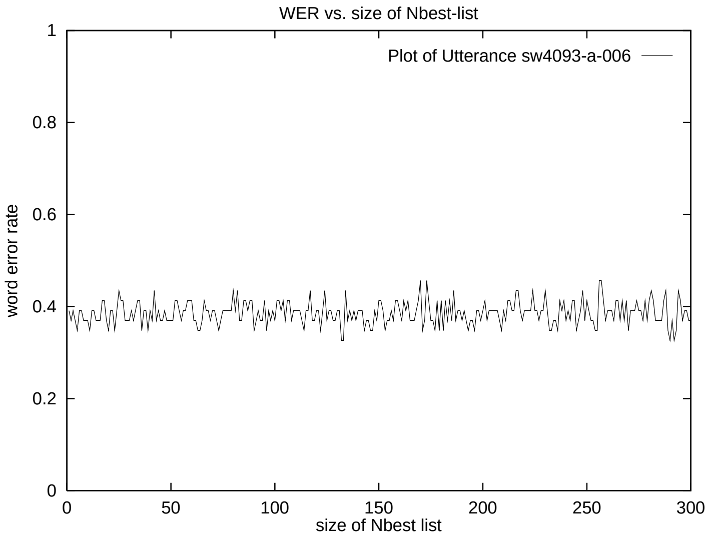


Figure 1.1: Example Graph for Erratic WER behavior of a hypothesis

# Chapter 2

# Review of Previous Related Work

## 2.1 Introduction

For a long time, computational linguists have focused on developing global coverage parsers and grammars. The idea is to be able to search through the whole potential search space of parse trees for any given input string and to resolve the ambiguities later by means of semantic, discourse, statistical, and possibly other knowledge sources.

However, for many practical applications where the input is not limited to a prespecified set of words and constructions, it became clear that there are many drawbacks which one faces with this theory-guided approach, such as incompleteness of lexicon, grammar, semantics; the issue of parsing time (usually $O(n^3)$ for $n$ words in the input string); the vastness of possible parses for even quite small sentences (parse ambiguity problem); failure on noisy or error-affected input (Abney, 1994; Grishman, 1995).

As a remedy to still be able to build reliable, robust, and fast parsers, several approaches of partial and/or shallow parsing have been proposed and developed which will be discussed below.

## 2.2 Differences between written and spoken language

Traditionally, most work in computational linguistics was mainly concerned with parsing written text. In part, this was due to the linguistic tradition, in part just to the unavailability of spoken language data. After the "Chomskian Revolution" around 1960, when the field of linguistics moved from a behaviorist to a mentalist perspective, competence grammars became the focus of interest and research. Particularly, it was *intentionally* abstracted away from all human factors such as attention span, memory capacity, production errors etc. (Chomsky, 1965). The basic question of linguistics became: why are some sentences and structures in a language grammatical while others are not? - and: What general (or: "universal") principles are there in the human language faculty that can provide an explanation for these facts? Discussed were (and still are) sentences like

> Which dog did the man with the telescope see a picture of? OR:
> The dog the mouse the cat chased hated ran away.

which may have yielded a great deal of theoretical insight, despite the fact that they are not always very representative examples of "naturally occurring" clauses.

But even when dealing with *written* language, it turned out that – whatever the linguistic theory of one's choice might be – writing grammars which account for all possible constructions in a language is a virtually impossible task: there are just too many of them to capture them all. And even *if* one gets to a close to perfect coverage, the issue of ambiguity beats back: sentences containing just a few words can lead to huge numbers of possible parse trees and this poses the problem of how to decide which one to pick.

As for spoken language, it has been recognized that a number of phenomena (purposely ignored by mentalist mainstream linguists) are prevalent and do pose a serious problem for almost all stages of language processing. To name a few (see e.g. (Lavie, 1996)): we find false starts, repairs, self-interruptions, repetitions, hesitations, stutters, filled pauses etc. A prototypical example of a segment of spoken language from the Switchboard corpus (Godfrey et al., 1992) is given here (also, to contrast it to the linguistic examples from above):[1]

> Well, you know, uh, talking about the lawyers, you know what might very well do, uh, cause a, uh, a drop in the number of lawyers and things like that, is to set the fees for cases. It's kind of like do it, do it in the similar vein similar, like, uh, V C R or television repair. If you take your T V in, a lot of these T V repair places will say, well, I'll repair your T V for a hundred dollars, and if he gets in there and starts rooting around and finds something in there that's really tremendously wrong with it, then he eats it.

## 2.3 Unification grammars

A very prominent approach in parsing both written and spoken language is the use of unification grammars and parsing algorithms which support these. The main underlying assumption is that every "unit", be it a word, (sub-) constituent, or phrase, bears a number of hierarchically organized features (or:

[1] Note that the punctuations are not found in the spoken source; they were inserted by the transcribers. Potentially, some of them may be found by silence or prosody detectors but certainly not all of them.

attributes) with associated values. Some values may be instantiated, others may not (yet) be so. In the process of compositionally combining the constituents – which is guided and licensed by the rules of the grammar – at each step the feature structures of the subcomponents are "unified" with each other. If unification fails due to inconsistent feature values, either the rule is not applied (strict system), conflicting values get marked as such or as "don't care" (relaxed system) or are resolved to the most plausible one (heuristic system).

An example of a unification driven parsing approach which successfully has been implemented both for written and spoken language is GLR/GLR* (Tomita, 1990; Lavie, 1996). In GLR*, the issue of ambiguity resolution is solved by incorporating probabilities to the actions in the LR parse table and using various heuristics from the parser and the discourse context (Lavie, 1996; Qu et al., 1996).

Within the German Verbmobil project (Wahlster, 1993), HPSG grammars (Pollard and Sag, 1994) which are also unification based are widely used by various research groups (Kasper et al., 1996; Kasper and Krieger, 1996; Kay et al., 1994).

In general, while these unification based grammars usually provide a fairly detailed analysis of the input string, they suffer from two problems: first, to deal with noisy and distorted input (particularly crucial in spoken language) and secondly, to be able to parse a string in real time (for many parsers, their time complexity is $O(n^3)$; further, the constants hidden in the $O$-notation are generally also not negligible.)

## 2.4 Link Grammar

Sleator and Temperley (1991) have developed a grammar formalism called *Link Grammar* which is based on the words in the lexicon and their *linking requirements*; links are labeled connectors which can attach to matching links to the right or left of the word. A sentence of the language defined by the grammar is a sequence of "correctly linked" words. The following three conditions have to be satisfied (Sleator and Temperley, 1991, 1):

1. Planarity: The links do not cross.
2. Connectivity: The links suffice to connect all the words of the sequence together.[2]
3. Satisfaction: The links satisfy the linking requirements of each word in the sequence.

Although the analyses obtained by this approach look similar to formalisms such as Dependency Grammar or Combinatory Categorial Grammar, a ma-

[2]This is relaxed in the version for the spoken language corpus Switchboard (Grinberg et al., 1995) with the possibility of skipping words in the input string, a similar idea to that in (Lavie, 1996).

jor difference is that Link Grammar does not have the notion of constituent-categories, i.e., it is entirely word-based.

A parsing algorithm with $O(n^3)$ runtime, enhanced by heuristics and massive pruning methods, has been developed for Link Grammars, and recently it was shown that it is quite feasible not only for written but also for spoken language parsing (Grinberg et al., 1995).

However, even though the parser shows reasonable coverage even for (noisy) spoken language, it suffers from three problems: (i) it assigns quite a lot of parse trees ("linkages") for a single short sentence[3]; (ii) unlike for unification based grammars, there is no obvious way how to get from the syntactic to the semantic structure of a sentence; and (iii) the large amount of link alternatives (in extreme cases more than 1000 for a single lexical entry) makes the lexicon rather difficult to understand and to maintain.

All in all, it appears that while Link Grammar might be a useful tool as a "grammar checker" (e.g. to assign scores of grammaticality to a list of speech recognizer hypotheses), it is unlikely that this approach can be usefully integrated in any higher-level module, such as semantic interpretation or information extraction.

An interesting study about using the Link Grammar approach for lattice (or Nbest list) rescoring was undertaken by (Jones, 1996). Jones used the cost vector from the Link Grammar to re-rank the top 1000 hypotheses from a speech recognizer lattice.[4] Over 86 utterances (some of which were split to sub-utterances for processing reasons), a small increase in word accuracy for the top-ranked hypotheses was achieved (0.3%), however this trend was somewhat inconsistent over different types of evaluations (e.g., different test corpora, varying size of Nbest lists).

## 2.5 Case frame parsing

It has long been noted that for applications in restricted domains, specifically for querying databases, the possible input strings are fairly regular and restricted. Secondly (and more importantly) it turns out that if one is not so much interested in a *complete parse*, but rather in the extraction of the *relevant information* for the given domain, a parsing approach which is based on case frames, i.e., semantic slots which can have certain kinds of fillers, is appropriate and efficient. A good example is the Phoenix system designed by Ward (Ward, 1991; Ward, 1994). The Phoenix system was developed for the ATIS[5] evaluations to facilitate the extraction of domain-relevant concepts in spontaneous speech where humans query a flight database (Ward, 1991; Ward, 1994). The main idea behind Phoenix is to use a semantic phrase grammar where semantic

[3](Grinberg et al., 1995) report 114 linkages on average for the Switchboard sentences which had each a maximum of 25 words.

[4]The utterances were from the ESST data collection at CMU (English Spontaneous Scheduling Task).

[5]Air Travel Information System

information is represented as a set of frames, which in turn consist of a sequence of slots. For each slot (which represents a "concept" in the domain), a separate grammar is defined which is compiled into a recursive transition network (RTN). The chart parser's task is to find the frame with a slot-sequence with the largest possible coverage over the input string. Heuristics such as "prefer flat trees over deep ones" and efficient search algorithms (beam search) are used additionally for ambiguity resolution and speed up. An example of the parser's output is given here (adapted from (Ward, 1994)):

```
input phrase:
 show flights from Boston to Denver after five pm
parsed phrase:
 [list] (show) [field] (flights)
 [from_loc] (from Boston)
 [to_loc] (to Denver)
 [depart_time] (after [start_time] ( [time] (five pm) ) )
```

Basically, Phoenix tries to *map* strings of the input phrase which are relevant to the domain to *concepts* which in turn can easily be used by a coupled interpretation module to get the intended "meaning" of the phrase.

Phoenix has meanwhile also successfully been integrated into the speech-to-speech translation engine JANUS (Waibel et al., 1996). The parser has proven to be more robust to the speech recognizer's output than the GLR* parser. Since the representation of concepts so far used in the system is less rich than it is for GLR*, the generation component which is based on the Phoenix parser's concepts, produces somewhat terser results than the GenKit generator (Tomita and Nyberg, 1988) based on the GLR* feature structures (Lavie, 1996).

The speed and robustness of Phoenix are the main reasons why I have decided to use this system also as a basis for my chunk parsing system; even with a very small, regular and unambiguous part-of-speech (POS) grammar, the coverage over Switchboard dialogue transcripts is very good. The ability to skip unparsable segments of the input makes it robust to ungrammaticalities of spontaneous speech, to speech recognizer errors[6], and to incomplete grammar coverage.

## 2.6 Statistical approaches

For more than a decade, some amount of research has been done in the areas of statistical inference of grammars and of statistical extensions to existing grammars. For the latter, we already mentioned GLR* as an example (Lavie, 1996) where statistical annotations in the grammar/parser facilitate the choice between ambiguous constructions (or in this case: "actions" in the parse table).

As for statistical inference, most approaches reported in the literature for automatically inducing probablistic context free grammars (PCFGs) have not

[6]This was shown in (Lavie, 1996) for the JANUS scheduling domain.

been as successful or promising as had been hoped for in the beginning (Lari and Young, 1990; Charniak, 1993). One lesson from this research is certainly that grammars tend to be somewhat "better" and "plausible" if they were induced from *bracketed* rather than unannotated corpora. A reason for this difference is that grammars induced from unannotated text tend to group frequently cooccurring tags together which are, on an intuitive level, not members of the same constituent — and therefore lead to unplausible and less useful generalizations.[7]

An early example of a probabilistic chart parser is the PEARL system (Magerman and Marcus, 1991). Recently, Brill (1995) reported experiments about rule-based induction of grammars which seems to be a more promising approach. (Vilain and Palmer, 1996) built on Brill's work, further refining and significantly speeding up the original algorithm, yielding to a parsing speed of more than 10000 words per second.

## 2.7 Finite state grammars

In contrast to the widely held belief that natural language is not regular (some say: not even context-free), there have been several attempts recently to develop parsing systems which operate under the assumption that regular expressions and finite state grammars can handle a significant and for most purposes sufficient amount of natural language input, specifically at the level of constituents. It is fair to mention that the time and efficiency constraints of some of the popular information extraction conferences (e.g., MUC[8], see (Grishman and Sundheim, 1996)) had a significant impact for recent developments and research in this direction (see also (Zechner, 1997)).

Pereira and Wright (1991; 1996) present an algorithm for computing finite-state approximations to context-free grammars which is exact for the subset of context-free grammars which generate regular languages, including right-linear and left-linear context-free grammars. While Pereira and Wright (1996) mention having used their method for language model construction for a limited domain speech recognition task, they unfortunately do not provide any results or details about that in their papers.

Koskenniemi (1990) uses a finite state syntax in his parsing and disambiguation system (for the languages Finnish, Swedish, and English). Each sentence is represented as a finite-state machine that accepts all possible readings of the sentence. Parameters for the interpretations are (i) word-form interpretation, (ii) clause boundary information, and (iii) syntactic tags (for each word). The grammar uses constraint rules for selecting the correct interpretations. These include (i) feasibility of clause bracketing, (ii) disambiguation rules for clause-types, (iii) clause boundary constraints, and (iv) constraints about the number of finite verbs. While this approach is obviously aimed at written (and hence fully grammatical) language, it is certainly interesting and promising for being

[7]A standard example is the very frequent sequence "of the" ("PREP DET" in POS notation) which is not as plausible as e.g. "the man" ("DET NOUN").

[8]Message Understanding Conference

applied to spoken language, as well. Unfortunately, no evaluation is made in Koskenniemi's paper, so nothing can be said about the overall performance of the system.

In the realm of the MUC-6 conference, MITRE (Vilain and Day, 1996) developed a rule-based finite-state phrase parsing system which gave very good results at the MUC-6 named entity blind test set (overall F-score[9] 91.2).[10] The system is based on Brill's (1994) approach of learning rule sequences. However, these results were obtained by *handcrafted* rules; a set of automatically created rules gave a somewhat worse performance (overall F-score 85.2). While these results are impressive in themselves, it has to be noted that this parser does not attempt a full input coverage but just concentrates on spotting certain types of phrases that are known in advance.

At SRI, the natural language research group (headed by Jerry Hobbs) has been developing the FASTUS system for about five years, which is also aimed at extracting information from natural-language text (Hobbs et al., 1992; Hobbs et al., 1996). FASTUS (in its current form (Hobbs et al., 1996)) consists of a five stage cascaded non-deterministic finite-state transducer. The stages are organized as a pipeline as follows: (i) recognition of fixed expressions (e.g., names), (ii) basic noun groups, verb groups, (iii) complex noun groups and verb groups, (iv) building of domain specific event-structures, (v) merging of event structures and database entry. FASTUS was probably the first large system built for a NL task which heavily relies on finite state technology. Its success and its inspiration for other approaches can hardly be overestimated. The main advantages of FASTUS are (i) its conceptual simplicity, (ii) its effectiveness, (iii) its fast run time behavior, and (iv) its fast development/adaption time when moving to a new domain.

Finally, I shall discuss Steven Abney's *chunk parsing* approach. Unlike the last two systems which are aimed at information extraction, his parser — called CASS — is of a general purpose type (Abney, 1990).

Abney's main idea is to implement a parser as a system of finite-state cascades (Abney, 1996b).[11] A finite-state cascade consists of a sequence of levels, the most important being the levels of *chunks* and *simplex clauses*. Chunks are contiguous, non-recursive cores of phrases like NPs, APs, etc., whereas simplex clauses are clauses where embedded clauses have been turned into siblings (i.e. iteration instead of recursion).

In the "Chunk Stylebook" (Abney, 1996a), Abney defines seven chunk categories: noun chunk (NX), verb chunk (VX), infinitive chunk (INF), present participle (or gerund) chunk (VGX), past participle chunk (VNX), adjective chunk (AX), and adverb chunk (RX).

Parsing itself is performed as a series of finite transductions, where each

---

[9] $F = \frac{(\beta^2+1)PR}{\beta^2 P+R}$, where $P$=precision, $R$=recall, $\beta$=a parameter which gives more weight to either precision or recall.

[10] The following phrase types had to be identified (SGML-tagged): organization, person, location, money, percent.

[11] In *that* sense, it is a similar approach to the one of (Hobbs et al., 1992; Hobbs et al., 1996).

transduction at each level is defined as a set of patterns. A pattern consists of a category symbol and a regular expression, hence it can be compiled into a finite-state automaton and therefore the parsing process is very efficient. Since there is no global optimization strategy involved, this cascaded parsing system is also very robust: the philosophy is "to do the easy things first" and to delay harder decisions (such as PP-attachments) as far as possible. Basically, Abney sees the parser proceed by "growing islands of certainty into larger and larger phrases" (Abney, 1996b, 2).

Abney (1996b) gives also some performance results for his parser: he reports a per-word chunk accuracy of about 92%[12] and precision/recall of about 88%/87%.[13] CASS (version 2) parses on the order of 1000-5000 words per second on a SunSparc workstation, depending on the number of levels involved.[14]

A version of the CASS parser was used for parsing and creating semantic annotations for Verbmobil data (Light, 1996). No evaluations of performance are given in this paper but it is (to my knowledge) the first attempt to integrate a finite-state parser with a spoken language system. Anecdotal evidence indicates that the parsing system is fast and robust but does not always provide the necessary information to create proper semantic representations. We take this as a strong hint that a semantic mapper has to be very error tolerant in order not to fail when faced with incomplete or inconsistent input.

## 2.8 Conclusion

There have been a large number of different approaches for parsing natural language. While some of them are derived from formal theories of language, such as theoretical linguistics, others are more guided by arguments such as speed and efficiency. Particularly for the task of information extraction, finite state parsers have been successfully implemented and employed. Since the chunk parsing technique has proven to be a general coverage, robust and fast architecture, it appears reasonable to use this technique for processing speech recognizer hypotheses.

[12] An independet human labeller got about 93% accuracy.

[13] 91%/88% for the independent human judge.

[14] Experiments with our Phoenix POS based parser show comparable results: the average parse speed is above 2000 tags/words per second.

# Chapter 3

# System Description

## 3.1 The General Picture

### 3.1.1 Global System Architecture

Figure 3.1 shows the global system architecture. The Nbest lists are generated from lattices that are produced by the JANUS speech recognizer (Waibel et al., 1996). First, the duplicates wrt. silence and noise words are removed, next the word stream is tagged with (Brill, 1994)'s POS tagger. Then, the token stream is "cleaned up" in the preprocessing pipe, which then serves as the input of the Phoenix POS based chunk parser. Finally, the chunk representations generated by the parser are used to compute scores which are the basis of the rescoring component that eventually generates a new reranked Nbest list.

### 3.1.2 Definitions

#### 3.1.2.1 Simplex Clause

A *simplex clause* serves as the basic unit for most of the main system components. I give its definition as follows[1]:

> A *simplex clause* is any finite clause that contains an inflected verbal form and a subject (or at least one of the two, if not possible otherwise). However common phrases such as *good bye*, *hello*, *thank you*, etc. are also considered simplex clauses.

This definition implies that all subordinate and relative clauses are split into separate units. This results in fairly small strings which are easier to handle for the system.

[1] I follow the definition of a *small clause* in (Gavaldà et al., 1997) which is different from the aforementioned definition of a *simplex clause* in (Abney, 1996b).

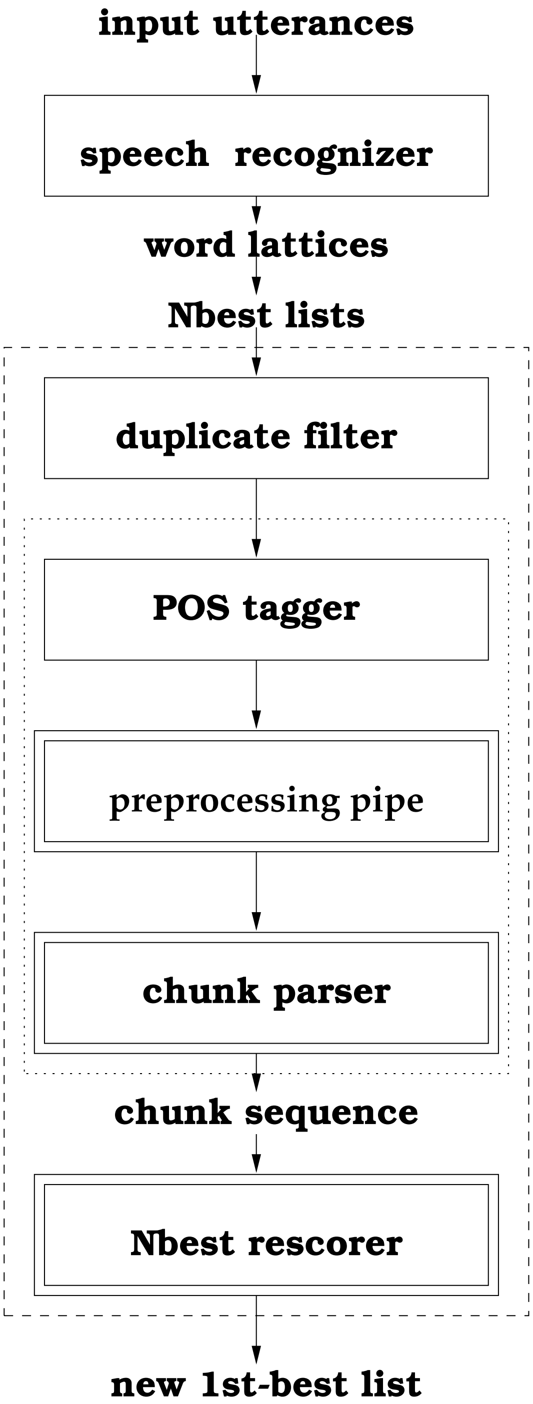


Figure 3.1: Global System Architecture

#### 3.1.2.2 Chunks

*Chunks* are contiguous non-recursive constituents within a simplex clause. Standard examples are: verb-groups, noun phrases, prepositional phrases etc.

The following example shows a break-up of a simplex clause into four such chunks, one per line:

```
[conjunction] (when)
[nounphrase]  (i)
[verbgroup]   (was interviewing)
[prepphrase]  (for a job)
```

### 3.1.3 Input

As a basis for this project, I mainly use the Switchboard (SWB) corpus database which consists of more than 2000 dialogues of spontaneous speech (approx. 3 million words in total) (see (Godfrey et al., 1992)).

As input for my system, I use SWB transcripts and speech recognizer hypotheses, extracted from Nbest lists (see the Appendix for an example). The latter come from a previous Switchboard evaluation (March 1996: Switchboard and Callhome databases).

### 3.1.4 Output

The output of the system is a first-best list which is derived from the re-ranked Nbest lists (for every utterance).

### 3.1.5 Resources

The Nbest lists are generated from word lattices produced by the JANUS speech recognizer (Waibel et al., 1996; Zhan et al., 1996).

The part of speech (POS) tagger was retrained for this task using the code and tools by Brill (Brill, 1994).

The reranking neural network is an adaptation of a standard neural net backpropagation package (Shufelt, 1994).

The chunk parser proper is the Phoenix parser (Ward, 1991) which so far has proven to be reliable and fast in the setting of ATIS, JANUS, ENTHUSIAST, and various related projects but is here — to my knowledege for the first time — used as a POS based parser.

## 3.2 System Components

### 3.2.1 Organization of this section

In this section, the modules that are used by my system are described in detail. First a *functional description*, the *module name* and the *programming language* in which it is written, are given. Next, the module's *input* and *output* behavior is characterized. Finally, in the case of modules where more than simple text processing is done, its *algorithm* is described. The description may optionally end with some additional *notes* that point out important considerations and/or observations.

### 3.2.2 Preparation of the Data

#### 3.2.2.1 Function

These modules take the Nbest-list, score it according to reference data files and split the data into a train set (for training/development) and an independent test set (for progress tracking). Particularly important are the annotations of word error rate (WER) for each hypothesis of each utterance in the Nbest-list.

Some modules are used only for the purpose of generating the train/test sets, whereas other modules (in **bold face**) also are used in evaluation mode, where the WER cannot be computed ahead of time.

#### 3.2.2.2 Splitting the Nbest list

**Programming Language** Perl

**Program Name** split.nbest.pl

**Function** Splits the Nbest list into N directories ("first-best" until "Nth-best") and writes dummy-hypotheses for all utterances having fewer than N hypotheses: in every directory we have a set of one hypothesis for each of the utterances.[2]

**Inputs** Nbest-list.
Format:

```
# utt-name start-frame global-recognizer-score hypothesis-rank
turn-label speaker start-frame duration hypothesized-word (or start/end/noise)
```

Example:

```
# en_4792_A-0000 7.73 2581.57 1
en_4792 A    7.73    0.04 (                     255.96  199.39
en_4792 A    7.77    0.12 YOU                   723.79  714.20
en_4792 A    7.89    0.24 KNOW                 1601.82 1561.00
en_4792 A    8.13    0.01 )                       0.00    0.00
```

**Outputs** In each of the n directories: a file hypo.n, in the same format, containing the n-th best hypotheses of all utterances.

[2]The N directories have to *exist* ahead of time. We need this splitting because of the scoring-module which assumes to have a list of one hypothesis per utterance, for all utterances.

#### 3.2.2.3 Scoring All Hypotheses

**Programming Languages** Tcl, Perl

**Program Names** score-all.pl
...invokes...
scoreSwb2.tcl
...and...
getscore.pl

**Function** Scores all N hypo.n-files (insertions, deletions, substitutions, correct words – with respect to reference files).

**Inputs** hypo.n-files (in directories 1...n)

**Outputs**

- hyp.ctm.pra.* files: detailed error-analysis
- hyp.ctm.sys: error-overview table
- scores.n files: format: utt-name, WER, correct words (C), substituted words (S), deleted words (D), inserted words (I)

**Algorithm** Main scoring script is scoreSwb2.tcl (by Torsten Zeppenfeld). getscore.pl just extracts the results from the .pra-files (simple string matching) and calculates the WER: WER=(S+D+I)/(C+S+D); if (C+S+D)=0: WER=0.

#### 3.2.2.4 NBest Reformatting and Duplicate Filter

**Programming Language** Perl

**Program Names**

- wer-annotate.pl
- **create-small-nbest.pl**
- **remove-dupl.pl**

**Function** These scripts generate a new Nbest format whose main advantage is compactness and so speeds up further processing. The original Nbest list is annotated with numbers for C/S/I/D (to be able to easliy compute the WER later on), duplicate hypotheses (which differ only in noise-words (ignored by our parsing/rescoring system)) are removed; in this process, important information about the various hypotheses and utterances is gathered and stored in files for the use by further system components.

**Inputs**

- scores.n files
- original Nbest list

**Outputs**

- annotated Nbest list (in #-lines, C/S/D/I-infos are added)
- Nbest list in new format (name: LABEL.data)
- Nbest list in new format without duplicates
- utterance file (see A.2; name: utts.LABEL; one hypo per line, only the words themselves)
- utterance based info file (see A.3; name: no-dup.LABEL.log)
- hypothesis based info file (see A.4; name: no-dup.LABEL.info)

**Algorithms**

- wer-annotate.pl: read in scores.i file and store C/S/D/I; read in Nbest-file and attach C/S/D/I-info in #-lines
- **create-small-nbest.pl**: just copy #-lines, for all other lines: just write the hypotesized words (no noises/start/end-symbols): s.t. we get one hypothesis per line now (instead of one word per line)
- **remove-dupl.pl**: read the Nbest list (in new format) and generate a new list which does not contain any duplicates (i.e., hypotheses that only differ in noise- and/or silence-words: these two word-types are irrelavant for our system); also store important information for other system components on some info-files (no-dup.*) and generate the utterance-file (utts.LABEL) which will be the input for further processing (i.e., the POS tagger component).

**Notes**

- The variable "LABEL" refers to the name of the working directory, i.e., where the Nbest list resides and the system was started. Many (but not all) files carry this LABEL in their name.
- The removal of duplicates has a very significant effect on the size of the Nbest list: whereas the average length of an Nbest list ($N = 300$) in the Devtest set is 232.8[3], the average length *after removing the duplicates* shrinks by about 64 percent to 82.7 hypotheses per utterance.

[3] For short utterances, there are less than 300 distinct hypotheses in the lattice.

#### 3.2.2.5 Generate train/test Data Sets

**Programming Language** Perl

**Program Name**

- split-data.pl
- gen-ref-cts.pl
- eval-score.pl

**Function** These modules split the data randomly in train and test sets, generate a count-file for the length of the reference-files (for normalization), and evaluate the WERs from the .data-files (generated by split-data.pl).

**Inputs**

- Nbest list in short format.
- one scores.i-file

**Outputs**

- *.data - files
- utts.ref.length
- wer.*.out- files

### 3.2.3 Part of Speech Tagger

#### 3.2.3.1 The POS tagger itself

**Function** The POS tagger assigns parts of speech (such as "noun", "adjective", etc.) to the words of the input string (i.e.: speech recognizer hypothesis). Basically, we use a version of (Brill, 1994)'s tagger (V1.14) which was adapted to and trained for the Switchboard corpus.

**Programming Language** C

**Program Name** tagger

**Inputs** A text file consisting of one utterance per line. All words have to be in lower case. No punctuation marks in the text.

Example:

```
you know if you only only percent of the for the for literature
was a good idea but tend to like your car now
```

**Outputs** The same text file annotated with POS (these POS are in upper case and follow a slash after the word).

Example:

`you/PRP know/VBP if/PREP you/PRP only/RB only/JJ percent/NN of/PREP the/DT for/PREP the/DT for/PREP literature/NN was/VBD a/DT good/JJ idea/NN but/CC tend/VBP to/TO like/PREP your/PRP$ car/NN now/RB`

**Note** Since the tagger "swallows" linefeeds on a random basis (roughly one per 1000 lines), a script was written to repair the output (repair-brill.pl).

**Tagset**

**Note** The tagset used is mainly based on slightly modified tagsets of the Penn Treebank and the Brown Corpus, as it appeared on a manually tagged subset of the SWB corpus. Some minor modifications were made for our system and are documented below.

```
* = not used by our system
! = new introduction
& = changed semantics from original tag set

! ANA anaphoric element, e.g. i can understand that/ANA
      (before usually marked as determiner); has often a noun-like
      behavior
! PRPA pers.pronoun in accusative case (no bound. before that)
! NEG negation particle (for semantics/info-extr.)
! AFF affirmative particle (... ditto...)
! CCC constituent conjunction (CC is ONLY a clause conjunction!)
      (no segm.boundaries before that)
! AUX-N negated AUX (isn't) (cheap, get for free)
! CV  conversational words (e.g. hi, bye-bye...; *could* also
      be tagged as RB, but get it quite cheap if restricted to few words)
! EOS pseudo-tag, only used by later components, not by POS-tagger itself

* ?? (not tagged??)
* . (end of sentence)
  AUX (might, can...)
& CC (conjunction), now ONLY clause type
  CD (card. number)
  DT (determ.)
  DT-AUX (that's)
  EX (there, here)
  EX-AUX (there's, here's...)
* FW (foreign word =>should be mapped to closest Engl. repres.)
```

```
* GW (unclear, e.g T/GW V/NN) -> letter-words (i/b/m...):  NNP in general
  JJ (adj.)
  JJR (adj.comp.)
  JJS (adj. sup.)
  NN (noun)
  NNP (proper noun)
* NNP.S. (probably typo)
  NNPS (plural proper noun)
  NNS (plural noun)
* PDT ("all", plural det.)
  PREP (preposition, can also take a gerund-(clause) as argument)
  PRP (pers.pronoun)
  PRP$ (poss. pers. pronoun)
  RB (adv. (modif.). e.g. always, just, kindof...)
  RBR (-"-, comp.)
  RBS (-"-, sup.)
  RP (verb-particle, e.g. set...up, walked...out/RP)
& TO ("to"+inf.), now all kinds of prep+gerund, as well
  UH (uh-huh etc.)
  VB (vb inf.)
  VBD (vb. past)
  VBG (vb. gerund)
  VBN (vb. past partic.)
  VBP (vb. present)
  VBZ (infl. verb 3rd sgl. present)
& WDT (rel. pronoun, but not consist. (that,which...)), now CONSISTENT rel.pr.
& WP (wh-particle, errors here, sometimes
      class. rel.pron. also "what", "who"...; =>
      now exclusively adjectival wh-particles (e.g. what company is ...)
  WRB (where/how/when: oblique argument-wh-part., standalone, if not
       a WDT)
* XX (mumbled word)
```

**Note:** Useful modules for the (re-)training procedure of the POS tagger are described in section 3.2.8.3.

### 3.2.4 Preprocessing Pipe

#### 3.2.4.1 Function

The preprocessing pipe (see Figure 3.2) is a sequence of PERL-scripts, taking the tagged hypotheses from the POS tagger as its input and producing a "cleaned up" version (for use by the chunk parser) as its output. Appendix B.1 illustrates a sample run through this preprocessing pipe.

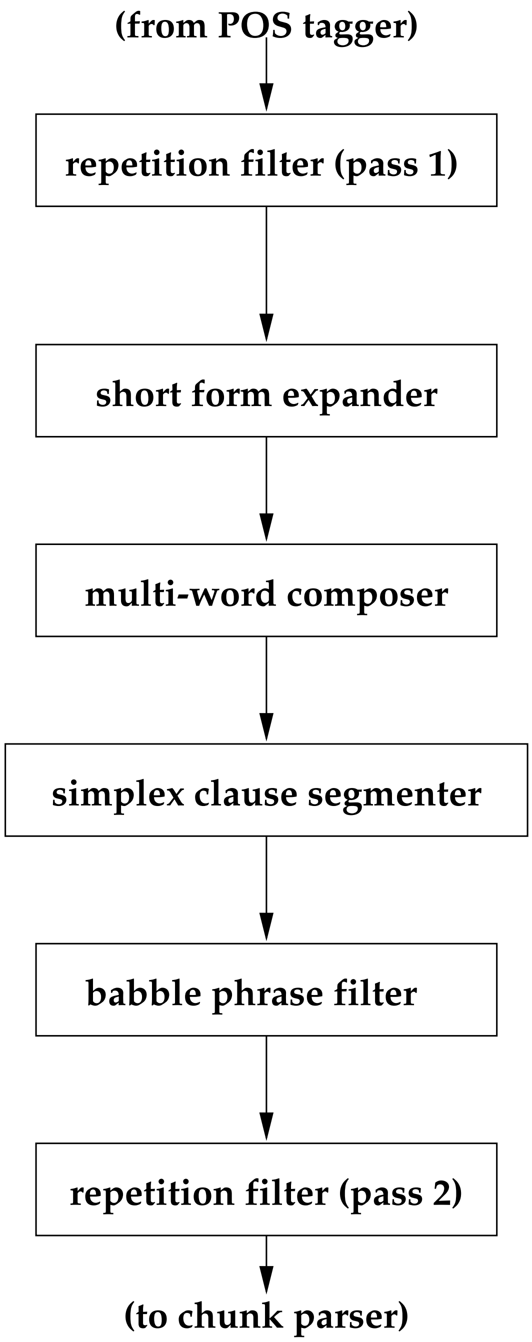


Figure 3.2: Preprocessing Pipe

#### 3.2.4.2 Repetition Filter

**Function** This filter eliminates repetitions of phrases up to length three; all repetitions found are reduced to the single occurrence.

It is used both right after the POS tagger and before the chunk parsing system component ("first pass"/"second pass"). The reason for two passes is that we want to keep the babble words/phrases for simplex clause segmentation, but once the babble filter has removed these, we also want to remove repetitions which had not been there before. (See the example below).

**Program** repfilter2.pl

**Programming Language** Perl

**Inputs** A text file with word/TAG-pairs.

Example:

`okay/UH i/PRP uh/UH i/PRP want/VBP to/TO i/PRP want/VBP to/TO talk/VB about/PREP these/DT these/DT issues/NNS`

**Outputs**

**First Pass:**

The same file with the removed repetitions (we still have babbles here):

Example:

`okay/UH i/PRP uh/UH i/PRP want/VBP to/TO talk/VB about/PREP these/DT issues/NNS`

**Second Pass:**

Meanwhile, we don't have babbles any more:

`i/PRP i/PRP want/VBP to/TO talk/VB about/PREP these/DT issues/NNS`

Now, we get...:

`i/PRP want/VBP to/TO talk/VB about/PREP these/DT issues/NNS`

**Note** It is not clear yet whether the POS-information should be ignored or not. Currently, it is *not* ignored which seems not to be a good idea in all cases. (Ex.: "...only/JJ only/RB..." does *not* get filtered now.) A good argmunent though for *not* ignoring the POS information would be cases like "he said that that house should be sold now" where the two "that"-tokens would get a different POS tag.

#### 3.2.4.3 Short Form Expander

**Programming Language** Perl

**Program Name** expand-contr.pl

**Function** Expands short forms (contractions) to facilitate the task of the chunk parser. (Examples: e.g. "it's, we're"...).

**Inputs** Text file with word/TAG pairs.

**Outputs** Text file with word/TAG pairs, the short forms being expanded.

**Algorithm** Simple string matching. However, with the "s"-form one has to ensure that it does not follow a noun (from the POS information) since this means we quite probably have a Saxon genetive form here.

#### 3.2.4.4 Multi-Word Composer

**Programming Language** Perl

**Program Name** multi-word.pl

**Function** Composes multi-words out of two or more single words to facilitate the task of the chunk parser. (Examples: because_of, a_lot_of, ...)

**Inputs** Text file with word/TAG pairs.

**Outputs** Text file with word/TAG pairs, with some multi-words.

**Algorithm** Simple string matching.

#### 3.2.4.5 Simplex Clause Segmenter

#### 3.2.4.6 Function

The simplex clause segmenter either runs as a neural net (NN) or as a rule-based Perl script. In the current final version of the system, the option for the NN cannot be used (the NN option needs more work and refinement; but the script performs about equally well, anyway.)

The NN takes a text file of word/TAG-pairs (one utterance per line) and generates (i.e. simply inserts) segmentation markers ("eos/EOS") in those places where it assumes that a simplex clause boundary might occur.

The training is based on a manually segmented file which contains "***" strings as "end of segment"-markers (Gavaldà et al., 1997).

The rule based script employs simple heuristics (e.g. "boundary before a conjunction or a non-accusative personal pronouns") to achieve the same task.

#### 3.2.4.7 NN in Segmenting Mode

**Programming Language** C

**Program Name** do_segm

**Function** Inserts hypothesized simplex clause boundaries ("eos/EOS") into a text-file consisting of word/TAG-pairs, one line per utterance.

**Inputs** Text file with word/TAG pairs.

**Outputs** Text file with word/TAG pairs **plus** inserted segment boundaries.

**Status** Currently, the input has to go conform with the format for the training program (Gavaldà et al., 1997). This should be changed to the here mentioned standard word/TAG-pair text-file format in future work. The main advantage would be a significant reduction in file size, probably without a loss in speed.

#### 3.2.4.8 NN in Training Mode

**Programming Language** C

**Program Name** NNsegment

**Function** Trains the NN on manually labelled data for the task of simplex clause boundary prediction, based on information about POS and trigger word in a small context window (for more information see (Gavaldà et al., 1997)).

#### 3.2.4.9 Segmentation Script

**Programming Language** Perl

**Program Name** ins-seos.ml.pl

**Function** Inserts hypothesized simplex clause boundaries ("eos/EOS") into a text-file consisting of word/TAG-pairs, one line per utterance.

**Inputs** Text file with word/TAG pairs.

**Outputs** Text file with word/TAG pairs **plus** inserted segment boundaries.

**Algorithm** Uses simple heuristics about where possible simplex clause boundaries can occur (e.g. before conjunctions, before personal pronouns which are non-accusative, etc.)

#### 3.2.4.10 Postprocessor

**Programming Language** Perl

**Program Name** ins-seos2.pl

**Function** Makes simplex clauses from utterances, using the boundary information: After each "eos/EOS" marker, a newline is inserted s.t. one utterance can now consist of multiple lines (one line per *simplex clause*).

Additionally, an index-file is created to keep track of the starting position of the hypotheses which are now consisting of multiple lines (one line per simplex clause).

**Inputs** Text file with word/TAG pairs (and eos/EOS markers).

**Outputs**

- Same text file with one simplex clause per line.
- Indexfile (see section A.5) (name: index.segm; lists the starting line for each hypothesis, containing possibly more than one simplex clause, starting with line 0)

#### 3.2.4.11 Babble Phrase Filter

**Function** Removes all babble words and phrases from the utterances.

**Program Name** remove-babb.pl

**Programming Language** Perl

**Inputs** A text file with word/TAG pairs, one line per utterance.

**Outputs** The same text file, but without babble words or phrases.

**Algorithm** Scan text file for all words with tag=UH and remove them from the text; also, remove some phrases wjoch are at least partially tagged with UH (e.g., "you/UH know/VBP"). If an utterance gets empty through this process, insert a dummy "eos/EOS" element to allow further components to still work with that utterance.

**Note** It is not clear how "far" this removal should go. Certainly, strings like "I guess" or "you know" can be considered babbles, but what about longer phrases like "and stuff like that"?

### 3.2.5 POS based Phoenix Grammar

#### 3.2.5.1 Function

The grammar is based on the POS tags used throughout our system. It is written using the Phoenix formalism which is based on frames and slots and requires one grammar-file (extension .gra) for each slot (Ward, 1991).

#### 3.2.5.2 Required Files

- forms (see A.6): this specifies all slots which can occur in a top level frame (we have only one frame in our grammar (called "chunks"))
- nets (see A.7): this specifies all slot names in the grammar (corresponding to a .gra-file)
- MAP_STRINGS (see A.8): this file is used by Phoenix as a preprocessing filter; in our system, we make no use of it, however
- Rule files (see A.9): these are the "proper" grammar rule files, one for each slot (and some include files for nonterminals used in more than one grammar file); nonterminals are either in square brackets (slot names) or start with upper case; terminals start with lower case. * or + markers in front of tokens indicate that this token may occur more than one time (*+: 0 or more times, +: 1 or more times, *: 0 or 1 times).

### 3.2.6 The POS Chunk Parser

**Function** This system component uses a POS based Phoenix grammar to parse the input string into chunks (in general: constituents like NPs, PPs, etc.) (see Figure 3.3). The parsing itself is done by the Phoenix parser (Ward, 1991).

The input to this component is a preprocessed file containing word/TAG-pairs, one line corresponding to one simplex clause.

The output is the parsed and formatted result file from the "Information Combiner" (see there).

#### 3.2.6.1 Word/TAG Splitter

**Programming Language** Perl

**Function** Splits word/TAG-pairs into two files of *just* words and tags, respectively.

**Program Name** splittag.pl

**Inputs** Text file with word/TAG pairs, one simplex clause per line.

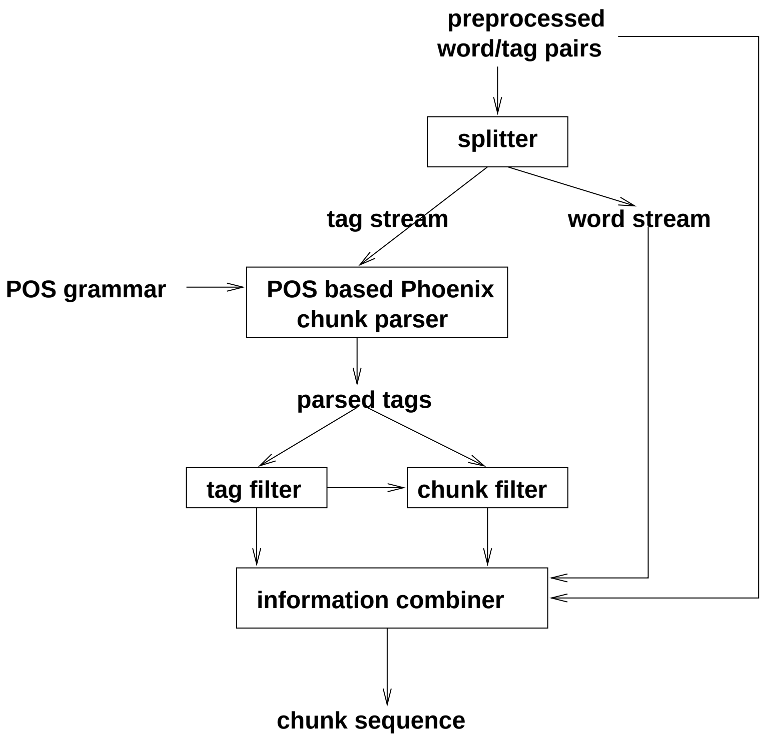


Figure 3.3: POS Chunk Parsing System

**Outputs** Two files: words.out (just containing the words), and tags.out (just containing the tags), same number of tokens per line.

**Algorithm** Trivial. (We need this script because the Phoenix POS grammar can take POS input exclusively.)

#### 3.2.6.2 Phoenix Chunk Parser

**Programming Language** shell scrpit, C

**Program Names**

- inter
- process_trans_ml

**Note** `inter` is a script which invokes the Phoenix parser (`process_trans_ml`).

**Function** Parses the input text (a POS sequence) according to the POS Phoenix grammar.

**Input** POS sequence, one line per simplex clause.

**Output** Parsed POS sequence, interspersed with various information from the Phoenix parser (for a sample output see section A.10).

**Algorithm** Phoenix Parser: See (Ward, 1991; Ward, 1994)

**Chunk labels**

```
Chunk Labels:
=============

(a) nominal
[_np]     noun-phrase
[_pp]     prep.phrase
[_whnp]   wh-noun-phrase (what experience do you have...)

(b) verbal
[_vb]     verb-complex (e.g. has been working)
[_vbneg]  verb-complex with negation (e.g. hasn't been working)
[_toinf]  to-infinitive (to+verb/prep+gerund)
[_vpart]  verb-particle (e.g. they moved out)
[_aux]    auxiliary verb (isolated)
[_auxneg] aux. verb, with negation (e.g. aren't you laerning...)
```

```
(c) adj/adv
[_adjp]   adjective phrase
[_adv]    adverbials

(d) connecting
[_conj]   conjunction
[_rpro]   relative pronoun

(e) pragmatic
[_expl]   expletives (there are...)
[_comm]   comment (that's...)
[_neg]    isolated negation (no...)
[_aff]    isolated affirmative (e.g. yes, yeah)

(f) other
[_wh]     wh-word (e.g. who is talking?)
[_misc]   rest-category (e.g. greetings: ``hello'')
[_eos]    end of simplex clause marker
```

Additionally to these "top level" chunk labels, there are three head-labels, for nominal, verbal, and prepositional heads. These are not exploited in the current version of the system but will be crucial in the next stage, when subcategorization information and selectional restrictions are applied.

#### 3.2.6.3 Parsed Tag Extractor

**Programming Language** Perl

**Program Name** ph-filter1.pl

**Function** Filters all lines from the Phoenix output file which start with a >: these indicate that some (possibly partial) parses were found and also indicate which tokens remained unparsed.

Additionally, a file containing the line-numbers of *non-parsed* simplex clauses is created. (Essential for the next extracting/combining components.)

**Inputs** Phoenix output file.

**Outputs**

- ph1.out (see section A.11): contains the POS lines which were (partially) parsed
- not.parsed.list: contains the line-numbers of not parsed simplex clauses

#### 3.2.6.4 Parsed Chunk Extractor

**Programming Language** Perl

**Program Name** ph-filter2.pl

**Function** Extracts a sequence of all parsed chunks from the Phoenix output file.

**Inputs**

- Phoenix output file.
- file not.parsed.list (from Parsed Tag Extractor)

**Outputs** File ph2.out (see section A.12): contains all parsed chunks, one chunk per line. A "delim"-line specifies the beginning of a new simplex clause.

**Remark** For this program to work, the file not.parsed.list must have been generated by the previous component (ph-filter1.pl) already. Therefore, these two filters cannot be run in parallel.

#### 3.2.6.5 Information Combiner

**Programming Language** Perl

**Program Name** combine.pl

**Function** Combines information from the previous extraction steps and assembles everything into a standard output format, on a simplex clause basis.

**Inputs**

- ph1.out (from the Parsed Tag Extractor)
- ph2.out (from the Parsed Chunk Extractor)
- not.parsed.list (generated from the Parsed Tag Extractor)
- segmented text-file *before* the Babble Filter
- words.out (words without tags, produced by splittag.pl)

**Outputs**

- Assembled chunk parse information in the following format:

```
(<== ... comment)

#569 <== number of simplex clause
{if/CC you/PRP only/RB the/DT only/JJ person/NN eos/EOS }
 <== original simplex clause, with babble words/phrases

[ _conj ] ( if/CC ) <== a parsed chunk
[ _np ] ( [ _pn_head ] ( you/PRP ) ) <== another parsed chunk
only/%JJ <== a non-parsed word
[ _np ] ( the/DT only/JJ [ _n_head ] ( person/NN ) )
[ _eos ] ( eos/EOS )
```

- chunks-line.out (see section A.13): This is a file containing all parsed chunks, one line per simplex clause, excluding "eos" chunks. It is needed for determining the chunk LM scores.

**Algorithm**

- read in not.parsed.list to get line-numbers of not parsed simplex clauses
- read in the tags-lines from ph1.out
- read in words.out (corresponding to the tags read in previous step)
- read in segmented.utts.out (segmented hypos, including babbles)
- while ph2.out is not EOF do
  - get line
  - split it into tokens
  - if (first token==delim) ⇒ we have a new simplex-clause
    - clean up unparsed tags in prev. clause
    - deal with a possibly unparsed sx-clause
    - initialize arrays for tags and words
  - else... (a parsed chunk)...
    - foreach token: combine the tags and words
- deal with remaining unparsed tags in current sx-clause
- deal with possibly remaining unparsed sx-clauses

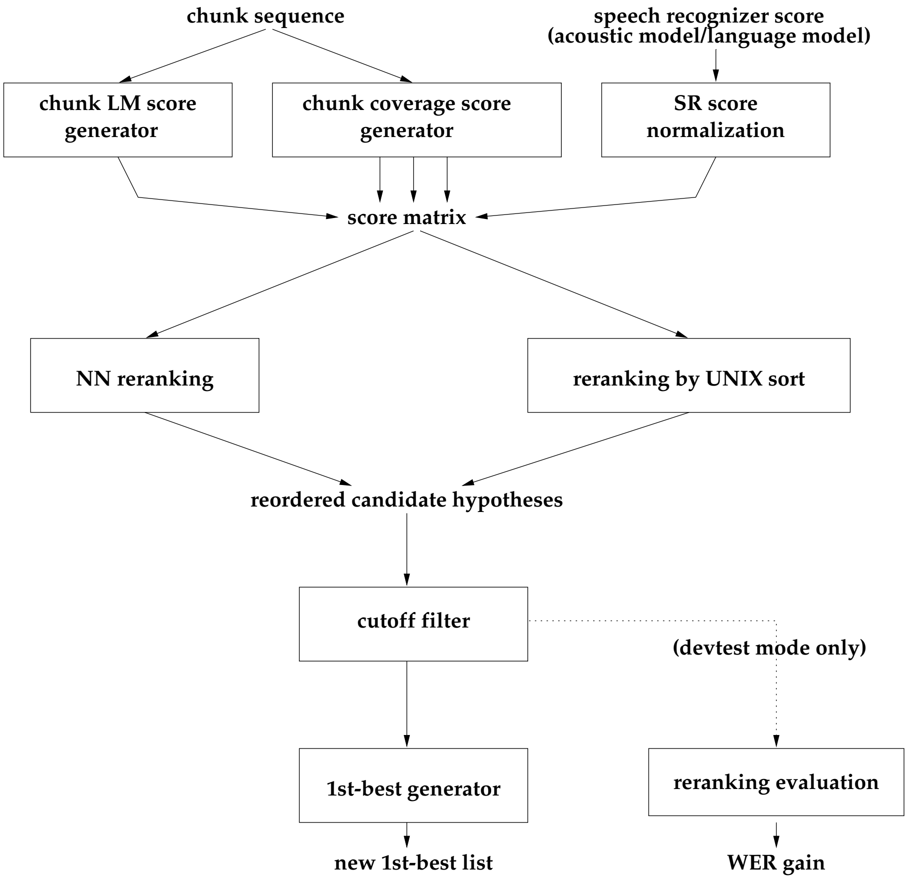


Figure 3.4: Nbest List Rescorer

### 3.2.7 NBest List Rescoring System

#### 3.2.7.1 Function

The rescorer (see Figure 3.4) consits of two main components: (i) the rescorer "proper" whose function it is to determine a new rank-order of the hypotheses, based on some scores calculated initally, and (ii) the reranking filter which decides based on some criterion/scores, which of the utterances to rerank and which not.

Both main components can be run using a neural net (NN) or a simple script. For the rescorer, the simple script just sorts the score-files for each utterance to some given (UNIX) sort command; for the filter, the simple script uses a cutoff value for the (average) hypothesis length: only utterances which exceed this length are eventually reranked. (In the current final version, there is only the option of using the simple cutoff filter; a NN filter could be plugged in, however, if properly trained resp. trainable.)

In "devtest" mode, the end of this component is a reranking evaluation where the average WER gain is computed; in "eval" mode (where WER gains cannot be computed anyway), the output is the new (reranked) first-best list, derived from the initial Nbest list which was given to the system in the beginning.

The various scores used here come from the speech recognizer and the chunk parsed output.

#### 3.2.7.2 Chunk LM Score Calculation

**Programming Language** C

**Program Name** wnric_new

**Function** Calculates chunk LM scores, for training/rescoring mode of the NN rescorer. Written by Klaus Ries.

**Inputs**

- dictionary file
- n-gram file
- chunks-line.out: chunk-sequences, one hypo per line, for chunk-LM-calculation

**Outputs**

- lm-scores.out: one score per line (i.e. per simplex-clause)

**Algorithm** The LM scoring is performed by Klaus Ries' program wnric_new.

#### 3.2.7.3 Chunk Score Calculation

**Programming Language** Perl

**Program Name** determ-score.pl

**Function** Calculates scores from the chunk parser's output (one set of scores per hypothesis), for training/rescoring mode of the NN rescorer.

**Parameters**

- working-directory ("LABEL")
- mainmode (devtest or eval)
- chunk-score-mode (segm_based/word_based)
- penalty-mode (word_skip_pen/segm_skip_pen)
- penalty factor (weighting factor for chunk LM penalty)

**Inputs**

- utts.LABEL.parser.out: chunk parsing system output file
- lm-scores.out: chunk LM scores (one per simplex clause)
- no-dup.LABEL.log: nr of hypotheses per utterance, utterance names
- no-dup.LABEL.info: various other scores (e.g. WER, normalized speech recognizer score)
- index.segm: starting line of each hypothesis (in LABEL.out we have *simplex clauses* as units, so we have to combine their scores to yield hypotheses-scores: all rescoring/reranking is based on *hypotheses*)

**Outputs**

- ./NNscores/scoreNN.utt-name: the generated score-files, one per utterance (contains all different hypotheses)
- LABEL.SC.out: chunk parsing system output files with annotations of scores, for devtest purposes

**Normalized Speech Recognizer Score** Since the global speech recognizer (SR) scores are frequently only minimally different within an Nbest list, we decided to use the `hypothesis rank` instead to reflect the speech recognizer's "opinion" about the relative "correctness" of the hypotheses that way. The formula for this rank-reflecting score is as follows:

$$norm\_SR\_score_{hypo_i} = \frac{size\_nbest - rank_{hypo_i} + 1}{size\_nbest} \tag{3.1}$$

**Chunk Coverage Score** The chunk coverage score should reflect how well the input was covered by the chunk parser. It is calculated as follows:[4]

$$chunk\_coverage\_score = \frac{num\_chunks}{elements} \tag{3.2}$$

where *elements* corresponds to

- number of words in the utterance **if** `chunk_score=word_based`
- sum of parsed chunks and non-parsed sections[5] **if** `chunk_score=segm_based` **and** `pen_mode=segm_based`
- sum of parsed chunks and non-parsed words **if** `chunk_score=segm_based` **and** `pen_mode=word_based`

To avoid a division by zero, $elements =_{def} 1$ in case of empty simplex clauses.

**Skipped Words/Sections Scores** These scores are "complements" of the chunk coverage scores and are calculated as follows:

$$skipped\_words\_score = \frac{num\_skipped\_words}{elements} \tag{3.3}$$

$$skipped\_sections\_score = \frac{num\_skipped\_sections}{elements} \tag{3.4}$$

where *elements* is defined exactly in the same way as for the chunk coverage score.

**Chunk Language Models**
In our system, two basic types of chunk language models were used:

1. (standard) backoff ngram models (trigrams and fivegrams)
2. models that penalize longer chunk sequences

**Chunk Ngram Models** The backoff ngram models were created as follows:

1. feeding the SWB corpus (3 million words) through our system, up to the chunk parser output, generating one chunk sequence for each simplex clause (there are over 500000 simplex clauses in the corpus)
2. postprocessing these chunk sequences generated by the module `combine.pl` s.t. they conform with the requirements of the next module
3. creating the ngram model using the `ngrammodel` tool (Ries et al., 1997)

[4]See section 3.3.5 for an explanation of the various parameters that are relevant here.
[5]A *non-parsed section* is a contiguous segment of non-parsed words.

We generated a trigram and a fivegram model, both get hitrates of the top ngram of over 99%; the perplexity is 5.0.[6] In the subsequent evaluations, these models made only a marginal difference in performance.

Since skipped words (or sections) were not included in this LM, we accounted for these by subtracting a weighted penalty; the system was tested on different weights and whether it would subtract this penalty for each skipped word or for each skipped section.[7]

The LM score for one hypothesis has to be a normalized combination of the scores of the simplex clause and is calculated as follows:

$$LM score = -\frac{\sum_{i=1}^{n} sx\_score_i - pen\_weight * skipped\_items}{length} \tag{3.5}$$

where $sx\_score$ is the ngram LM score for a simplex clause[8], $length$ is either the number of sections or the number of words in the hypothesis (system parameter `chunk_score`), $pen\_weight$ was varied between 0.0 and 2.0, and $skipped\_items$ could, as mentioned above, be either the number of skipped words or the number of skipped sections.[9]

**Length Penalty Chunk Language Model** Since we made the observation that a (comparatively) high number of chunks within a simplex clause usually means that the input is likely to be ill-formed (i.e., the chunks are *shorter* than on average), we are using a second chunk LM here, which exactly accounts for this fact.

It is derived as follows:

$$LMlength\_score_i = \frac{1.05^{num\_chunks}}{num\_chunks} \tag{3.6}$$

which means that the longer the simplex clause, the higher the score will be.

The total score for a hypothesis is calculated exactly in the same way as in equation 3.5, substituting $LMlength\_score$ for $sx\_score$.

#### 3.2.7.4 Data Preparation for NN Rescorer

**Programming Language** Perl

**Program Name** prep-rescore.pl

**Function** Reads score-files produced so far to generate the appropriate input format for the NN rescorer. Basically, all columns after a preindicated number are reproduced to STDOUT. (Should be directed to: ALL.scores.NN)

[6] We have only 19 different chunk types (the `eos` chunk is ignored throughout scoring).

[7] A *section* is either a chunk or a contiguous segment of non-parsed words in the output of the chunk parser.

[8] This is computed by Ries' utility `wnric_new`.

[9] Since we are using the *negative* LM score, a *low* score indicates a high probability for a given chunk sequence.

**Inputs**

- no-dup.LABEL.log (for utterance names)
- ./NNscores/scoreNN.*-files

**Outputs**

- ALL.scores.NN (see section A.14) (from STDOUT): one file, containing the scores used by the NN rescrorer; first column has to contain WER (in case of eval-mode: dummy-0.0-values)

#### 3.2.7.5 NN Rescorer in Train Mode

**Programming Language** C

**Program Name** NNrescore

**Function** NN for estimation of true WER for each hypothesis (input: various scores, target: true WER).

**Inputs** ALL.scores.NN: Vector-File: first column=true WER, all other columns: various scores.

**Outputs** Trained NN. (Weights-File)

**Algorithm** Standard Backpropagation (backpropagation part implemented by Jeff Shufelt; code from Tom Mitchell's Machine Learning class, fall term 1996, (Shufelt, 1994)).

#### 3.2.7.6 NN Rescorer in Rescore Mode

**Programming Language** C

**Program Name** do_rescore

**Function** Estimates WER from various scores.

**Inputs**

- Trained NN
- ALL.scores.NN: same vector file as for NNrescore. But obviously, the first column is ignored (true WER); in evaluation mode, we fill that with a dummy-value (e.g. 0.0).

**Outputs** File with two values per line: first-column-value (true WER in devtest-mode), estimated WER.

**Algorithm** Simple feedforward of the input, using the weights from the previously trained NN.

#### 3.2.7.7 NN Rescorer Postprocessing

**Programming Language** Perl

**Program Names**

- divide-resc-res.pl
- comb-resc-info.pl

**Function** Dividing the sorted results of the rescorer NN into the directory ./resc-res, producing one file per utterance (divide-resc-res.pl); putting some more info to these files, derived from previously created info-files (comb-resc-res.pl).

**Inputs** (*Note: LABEL=current working directory*)

- no-dup.LABEL.log: nr of hypotheses per utterance
- NN result file (true and estimated WER)

**Outputs**

- ./resc-res/utt-name.sorted: rescorer-result-files (three values per line)
- ./resc-res/utt-name.NNrerank.sorted: comprehensive infos about each hypothesis of each utterance (the estimated WER and the hypo-nr appear at the end of the lines)

#### 3.2.7.8 Simple Rescorer

**Programming Language** Perl

**Program Name** simple-reramk.pl

**Function** Reranks the score*-files in ./NNscores by simply sorting these using a UNIX sort string command (e.g. "-k 5n -k 6rn -k 4rn").

**Inputs**

- no-dup.LABEL.log: utterance-names
- scoreNN.*-files in NNscores

**Outputs**

- ./resc-res/utt-name.simple-rerank.sorted

**Note** Before sorting, intermediate files are created which have dummy-WERs and line-numbers appended to each line, for compatibility with the output from the NN rescorer. (./NNscores/scoreNN.plus2.*-files)

#### 3.2.7.9 Data Preparation for Rerank Filter

**Programming Language** Perl

**Program Name** prep-dec-resc.pl

**Function** Extracts vectors from the summary-file generated by eval-rerank.pl to produce a suitable input file for NN training/testing.

**Parameters**

- ranking method (NNrerank/simple-rerank)
- working directory
- "train" (optional: if specified, all.summary is read and the WER gain/loss information is extracted to column 1, for NN training; else: a dummy value (0.0) is put to this column)

**Inputs**

- ./resc-res/*.sorted
- all.summary (from eval-rerank.pl, if flag "train" is specified)

**Outputs** ALL.scores.filter (column 1: WER gain/loss (or 0.0))

**Notes**

- The WER gain/loss info has to be in column 2 of all.summary. Currently, the extracted columns and their weight-factors are fixed; we could implement these later as parameters.
- In the current version of the system, the rerank method **has** to be specified as `simple-rerank` since the NN rerank filter has not been fully developed.

#### 3.2.7.10 Reranking Filter in Train Mode

**Programming Language** C

**Program Name** NNdecision

**Function** Tries to determine when to rerank and when not to: If the target is negative (delta $< 0$: the reranking put a worse hypo to the top), we don't want this utterance to be reranked and if positive, we do.

**Inputs** Vector file; first vector: delta (old_WER - new_WER), the other vectors: several scores and features.

**Outputs** A trained NN (weights).

**Algorithm** Standard backpropagation NN (the backpropagation routines were developed by Jeff Shufelt, code from Tom Mitchell's Machine Learning class, fall 1996 (Shufelt, 1994)).

#### 3.2.7.11 Reranking Filter in Filter Mode

**Programming Language** C

**Program Name** do_dec

**Function** Decides whether to rerank an utterance or not, based on the previously trained NN.

**Inputs** Trained NN; vector file is same as for training, but obviously, the first column is ignored; in evaluation-mode we have to fill in a dummy-value (e.g. 0.0).

**Outputs** For each utterance: Either 0 (don't rerank) or 1 (rerank), one integer per line.

**Algorithm** Simple NN feedforward of the input.

#### 3.2.7.12 Simple Filter

**Note** in the current final version of the system, this is the only option for filtering.

**Programming Language** Perl

**Program Name** simple-filter.pl

**Function** Decides whether to rerank an utterance or not, based on the average length of the hypotheses of the current utterance.

**Inputs** ALL.scores.filter (contains average hypothesis length in column 2)

**Outputs** For each utterance: Either 0 (don't rerank) or 1 (rerank), one integer per line.

#### 3.2.7.13 Reranking Evaluation

**Programming Language** Perl

**Program Name** eval-rerank.pl

**Function** Reranks the hypotheses for each utterance according to the estimated WER from the NN rescorer. Writes evaluation files which tell about the difference in WER between old and new Nbest list.

**Inputs**

- filter.out (from NN filter/simple filter)
- no-dup.LABEL.log
- ./resc-res/*.sorted-files

**Outputs**

- ./resc-res/utt-name.eval: for each hypothesis: comparison between this position in old vs. new Nbest list (delta-WER, average-WER old/new)
- all.summary: summary information for all utterances (e.g., first ranked hypothesis number, average WER gain)

**Note** The position of the info in the *.sorted-files is crucial for this program.

#### 3.2.7.14 First Best List Generator

**Programming Language** Perl

**Program Name** make-new-1stbest.pl

**Function** Generates first-best list based on the rerank-filter info in filter.out and on the info in extract.info (is generated at run-time). Uses the original Nbest list for extracting (s.t. we get the same format in the end).

**Parameters**

- working-directory
- reranking method (NNrerank/simple-rerank)

**Inputs**

- filter.out (0 or 1: no rerank/rerank)
- no-dup.LABEL.out: utterance names
- ./resc-res/*.sorted
- original Nbest list (from STDIN)

**Outputs**

- First-Best List (to STDOUT; one hypo per utterance, same file format as Nbest list)
- extract.info (see section A.15) (utt-name, nr of best hypo found in rescorer)

**Note** This program assumes that the original hypo-nr is in the first column of the *.sorted-files in ./resc-res.

### 3.2.8 Miscellaneous Modules

#### 3.2.8.1 Function

Various useful modules/programs/scripts which are not directly parts of the system itself.

#### 3.2.8.2 Computing the WER curve

**Programming Language** Perl

**Program Name** comb-score.pl

**Function** Computes the theoretically best WER achievable up to a size k of the original Nbest list, i.e., assuming we knew which hypothesis we have to pick (i.e., the one with the lowest WER so far).

**Inputs** scores.n files

**Outputs** WER files, for gnuplot (two values per line: index k, best WER so far)

#### 3.2.8.3 Modules for Training and Testing of the POS tagger

**Function** (Brill, 1994)'s tagger (V1.14) comes along with various utility scripts for training and adaptation, but there is no "global" training script available. With the guidance of the different README files, I constructed some useful scripts that greatly facilitate these essential tasks of (re-)training, adaptation, and testing of the tagger.

**Program Names**

- doprep.pl
- pot-tag.pl
- dotrain.pl
- merge-lex.pl
- eval-tagger.pl

**Programming Language** Perl

**Inputs**

- doprep.pl: The input is initially an untagged corpus. The file-lines, where to start and where to end tagging are also given.
- pot-tag.pl: A tagged (sub-)corpus.
- dotrain.pl: Two tagged (sub-)corpora
- merge-lex.pl: a small lexicon with tag frequencies from the current training corpus and a large lexicon (the "original" one)
- eval-tagger.pl: A correctly tagged reference (sub-)corpus and an automatically tagged equivalent (sub-)corpus

**Outputs**

- doprep.pl: A pre-tagged version of an untagged corpus, useable for manual tagging
- pot-tag.pl: A tagged (sub-)corpus with annotations for potential tagger-errors
- dotrain.pl: updated files for "contextual rules" (and possibly also for "lexical rules" and for the tagger-lexicon)
- merge-lex.pl: a new lexicon, where the order of tags may have changed
- eval-tagger.pl: log-files with error-statistics and global accuracy evaluation (.log, .stat, .unswords: three output-files with decreasing amount of error-information)

**Algorithms**

- doprep.pl: Tagging of the untagged subcorpus with the current version of the tagger; invoking pot-tag.pl for attachment of tag-alternatives.
- pot-tag.pl: For all tag/word-combinations which are frequently mistagged, tag-alternatives are added for ease of manual editing.
- dotrain.pl: (re-)trains the tagger, using the first half of the data for improving on the "lexical rules" (optional) and the lexicon (also optional), and the second half for an improvement on the contextual rules; finally, eval-tagger.pl is invoked for evaluation of current accuracy
- merge-lex.pl: re-orders the tags in the big lexicon if they are in a different order in the small lexicon (order is representing frequency)
- eval-tagger.pl: comparison of a manually tagged model-file with a machine-tagged file, producing various sorts of statistics (see above)

**Note 1:** To perform the training, the following steps are usually done:

1. work on the lexicon, prepare as much as possible ahead of time
2. get the untagged source corpus
3. repeat the following steps, until the whole corpus is tagged
   (a) call doprep.pl with 2 indices corresponding to the lines of the current subcorpus you are working with
   (b) manually correct the tagging using the output of that module
   (c) split the total corpus that has been tagged so far in two halves and call dotrain.pl using these two files as arguments

**Note 2:** Since speech recognizers work with a fixed dictionary, it does not make much sense in training the lexical rules since they are meant for predicting the correct tags for *unseen words*. Likewise, the modification of the lexicon should not be necessary during the retraining procedure, unless one encounters words where the tag frequencies do not correspond to the ordering in the original lexicon. In these case, the "lexicon-merge-option" can be used in dotrain.pl, but special care is advisable when doing that.

#### 3.2.8.4 Correlation Modules

**Programming Language** Perl

**Program Names**

- correl.pl
- multi-correl.pl

**Function** Calculates Pearson r correlation coefficient between vectors of numbers. The first vector is the "reference" (column specified as parameter), all the others *following* this one, will be correlated with that. (multi-correl.pl can process a *list* of files, invokes correl.pl)

**Inputs** Vector file. N rows, M columns (real numbers)

**Outputs** Pearson r for every correlation pair.

#### 3.2.8.5 Default Config File Creation

**Programming Language** Perl

**Program Name** mk-def-config.pl

**Function** Creates default configuration file for run-all.pl, by reading all its default values.

**Inputs** run-all.pl (from STDIN)

**Outputs** default.config (to STDOUT)

## 3.3 Running the System

### 3.3.1 Hardware and Environment

While the system in principle is architecture independent[10], some of them currently are only compiled for the Alpha architecture. Therefore, to run the whole system, it has to be run on an Alpha machine.

The various components reside under the Interact Labs /net-environment, which forms the background environment for our system.

### 3.3.2 Preparations

It is advisable to create an empty directory where one puts only the whole Nbest list in. Alternatively, if one wants to run the system on multiple machines, one can split the Nbest-list into k pieces (each one **has** to start with a new utterance!), dirstibute them into k separate directories and then run k processes on N machines simultaneously.

[10]All parts are either coded in C or in PERL

### 3.3.3 Main Executing Script

The main executing script is run-all.pl. It has to be called with the following command line parameters:

- name of current directory (required)[11]
- name of the Nbest list file (required)
- name of the configuration file used (optional; if none is specified, the system runs with the default settings in the script run-all.pl, see below)

More parameters can be specified, in the following format:
`-parameter_name 'parameter_value'`
e.g., `-cutoff '12'`
where `parameter_name` has to correspond exactly to the name in the configuration-file (resp. in the script run-all.pl) and the parameter should appear inside of single quotes (this is *necessary* if it consists of more than one string, e.g.: `-rerank_sortstr '-k 5n -k 6rn -k 4rn'`)

While the script is running, it produces log information both to the screen and appends this information also to a log-file (run-all.log).

The final output files are:

- in `devtest` mode: all.summary (information about each utterance, specifically about the new top ranked hypothesis, and the average WER gain)
- in `eval` mode: the new first-best list, created based on the reranking results in the format of the original Nbest-list (filename: Nbestlist.out, where Nbestlist was the inputfile for the system)

### 3.3.4 Runtime

Approximate system runtime (from the POS-tagger to the end of the executing script) for 103 utterances (ca. 8500 hypos) on an empty Alpha (200 MHz, 192MB RAM) is less than 10 minutes, i.e. the throughput is more than 10 utterances/minute.

### 3.3.5 Configuration File and System Paramters

The following is the default configuration file which can be created automatically using the command:
`mk-def-config.pl < run-all.pl > default.config`

*Note: some newlines were inserted here for readability, they are marked with* \\.

Table 3.1 explains the parameters and their possible range of values.

[11] It has to be the *dircetory name*, *not* the full path name!

```
# Automatically created default.config-file
      num_nbest : 300
      main_mode : devtest
      demo_mode : no
        cleanup : no
         cutoff : 8
    start_label : BEGINNING
     exit_label : END_OF_SYSTEM
     tagger_dir : /net/tink/usr4/zechner/brill-new/New-Tagger-SWB
     tagger_cmd : /tagger LEXICON.NEW.SWB %s BIGRAMS LEXRULEOUTFILE \\
                    CONTEXT-RULEFILE
    segm_method : script
  rerank_method : nn
 rerank_sortstr : -k 5n -k 6rn -k 4rn
  filter_method : simple
    NN_segm_cmd : /do_segm -n h2t0.7w8.17Jan.net -c 69 \\
                    -T -2 segmented.utts.witheos -O 0.7 -w 8 -h 2
  NN_rerank_cmd : /do_rescore \\
     -n /net/tink/usr4/zechner/bin/rescorer/mar26.dev.c0.h1.e10.net \\
     -c 6 -T -2 ALL.scores.NN.out -h 1
  NN_filter_cmd : /do_dec -n h2t0.5mXX0000.17JAan.net \\
     -c 6 -T -2 filter.out -O 0.5 -h 2
       dir_segm : /net/tink/usr3/zechner/nn-segmentAL
     dir_rerank : /net/tink/usr4/zechner/bin/rescorer
     dir_filter : /net/tink/usr3/zechner/dec-resc
POS_grammar_dir : /net/tink/usr3/zechner/nlp-3/grammars/std-Mar21
    Phoenix_dir : /net/tink/usr3/zechner/chunker
    Phoenix_cmd : process_trans_ml -debug 1 -CHECK_AMBIG 1 \\
      -PROFILE 1 -PRINT_BEAM 1 -ignore_oov 0 -add_new_rules 0 \\
      -START_END 0 -STRIP_NT 0 -PRINT_INTP 1 -grammar %s
       perl_dir : /net/tink/usr3/zechner/nlp-3/perl-scripts
    chunk_score : word_based
     pen_factor : 1.0
       pen_mode : word_skip_pen
   lm_score_dir : /net/tink/usr4/zechner/new-LM
  lm_score_prog : wnric_new
      readlines : -S
      lm_arpabo : /net/tink/usr4/zechner/new-LM/p027-std.3.arpabo.3
        lm_dict : /net/tink/usr4/zechner/new-LM/p027-std.3.vocab
     wer_column : 3
   nbest_cutoff : 50
```

| Parameter Name | Description | Range of Values |
|---|---|---|
| num_nbest | (max.) size of Nbest list | `integer` |
| main_mode | "development"/"evaluation" | `devtest/eval` |
| demo_mode | "yes": screen-output; writes demo.log; proceeds in step-mode | yes/no |
| cleanup | "yes": deletes most intermediate files | yes/no |
| cutoff | only utts. with avg. length < cutoff are reranked | `integer` |
| start_label | where the script starts | `script-label`[a] |
| exit_label | script stops *before* here | `script-label`[a] |
| tagger_dir | `tagger`/other POS resources | `full pathname` |
| tagger_cmd | command to invoke tagger | see default-file |
| segm_method | type of segmentation method | script/nn[b] |
| rerank_method | type of reranking method | simple/nn |
| rerank_sortstr | UNIX sort string for reranking | a valid UNIX sort string |
| filter_method | type of filter method | simple/nn[b] |
| NN_segm_cmd | invoking the NN segmenter | see default-file[c] |
| NN_rerank_cmd | invoking the NN reranker | `-n net_name -c nr_inputs -T -2 output_file -h nr_hiddens` |
| NN_filter_cmd | invoking the NN filter | see default-file[c] |
| dir_segm | location of NN segmenter | see default-file[c] |
| dir_rerank | location of NN reranker | see default-file |
| dir_filter | location of NN filter | see default-file[c] |
| POS_grammar_dir | location of POS grammar | see default-file |
| Phoenix_dir | loaction of Phoenix executable | see default-file |
| Phoenix_cmd | invoking Phoenix | see default-file |
| perl_dir | location of system Perl scripts | see default-file |
| chunk_score | for chunk score calc. | `word_based/segm_based` |
| pen_factor | for chunk LM scores | `real` |
| pen_mode | type of skipping penalty | `word_skip_pen/segm_skip_pen` |
| lm_score_dir | loc. of `wnric_new` | see default-file |
| lm_score_prog | chunk scoring program | `wnric_new` |
| readlines | parameter for `wnric_new` for processing of multi-word lines | `-S` |
| lm_arpabo | trigram table file (gnuzipped) | `full path name` (without suffix `.gz` |
| lm_dict | chunk LM dictionary | `full path name` |
| wer_column | column of score-file which contains the WER | 3 |
| nbest_cutoff | only hypotheses that are before this cutoff in the original Nbest list are moved to the front | `integer` |

[a]Valid script-labels are: BEGINNING NBEST_PREP TAGGER REP_FILTER_1 SEGMENTING BABBLE REP_FILTER_2 CHK_PARSER LM_SCORES RERANK FILTER MAKE_FIRST_BEST END_OF_SYSTEM

[b]The option `nn` is currently not available.

[c]Since option `nn` is unavailable, this parameter has currently no effect.

Table 3.1: Explanation of System Parameters

# Chapter 4

# Evaluation

## 4.1 Definition of Notions and Metrics

Before I am giving the details of the system evaluation, I will define some important notions and metrics that I will be using below.

- **Word Error Rate (WER)**: $\frac{subst+del+ins}{correct+subst+del}$;[1] the denominator corresponds to the length of the reference hypothesis.[2]
- **True WER**: The WER that is calculated from the first best hypotheses from the original Nbest list ("what we get, if we don't do any rescoring")
- **Optimal WER**: The WER that would be achievable, if we were picking the hypothesis with the lowest WER from each Nbest list ("cheating experiment")
- **WER gain**: `previous_WER - new_WER`: the difference in WER before and after reranking (comparing the first best hypotheses); this value is *positive*, if the WER *decreases* due to the reranking process
- **Word Gain**: `WER_gain * reference_length`; this is a particularly useful metric when comparing two utterances of different length: it says "how many more words" one could get right within a (reranked) Nbest list of a single utterance.
- **Expected WER Gain**: WER gain in case of a randomized selection of a hypothesis from the Nbest list ("baseline case")

[1] $subst$ = word substitutions, $ins$ = word insertions, $del$ = word deletions.

[2] If the reference hypothesis has zero length, the WER is defined to be equal to 0.0.

| Data Source | Utterances | maximal WER gain |
|---|---|---|
| Switchboard | 154 | 9.7% |
| Callhome | 220 | 16.9% |
| Total | 374 | 13.1% |

Table 4.1: General Properties of the Devtest Set

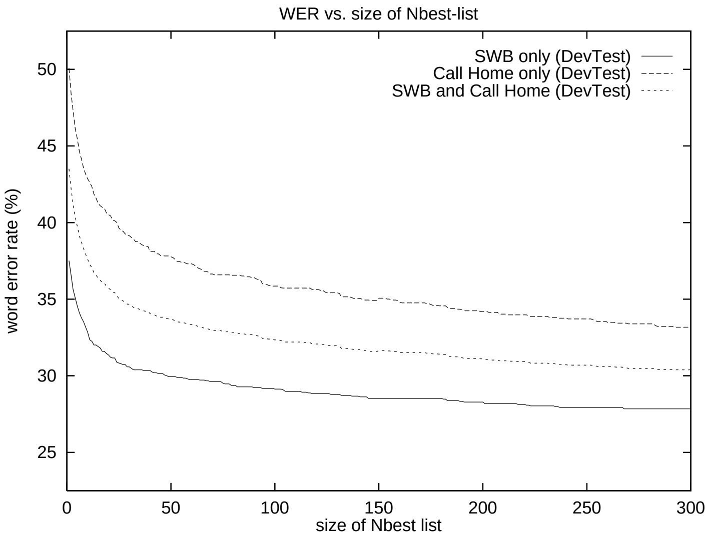


Figure 4.1: Potential decrease in WER over size of Nbest list

## 4.2 Properties of the Data

### 4.2.1 Data Used for General System Development

For general system development, i.e., training of the neural network, testing of various system parameters, and stepwise refinement and improvement of the system components, a subset of the March 1996 Switchboard Evaluation Data was used, comprising 374 utterances in total, partly from Switchboard, partly from Callhome data.

The properties of this data are given in Table 4.1; Figure 4.1 shows the potential decrease in WER if one knew which hypothesis to rank first over the length of the Nbest list for the SWB and Callhome data subsets and the overall Devtest set.

Table 4.2 shows the distribution of the word gain, together with other parameters, such as average WER on the first-best hypothesis, average length of the hypotheses, etc.

For the purpose of training and testing the neural net rescorer, the data

| word gain | total weighted word gain | avg. index of best hypo | avg. WER of first hypo | avg. length of hypos | total nr. of hypos (in %) |
|---|---|---|---|---|---|
| 0 | 0.222 | 3.070 | 0.197 | 3.725 | 142 ( 37.97) |
| 1 | 94.460 | 33.739 | 0.366 | 10.641 | 92 ( 24.60) |
| 2 | 114.954 | 70.386 | 0.446 | 14.474 | 57 ( 15.24) |
| 3 | 139.405 | 126.130 | 0.512 | 18.217 | 46 ( 12.30) |
| 4 | 84.778 | 118.095 | 0.563 | 20.286 | 21 ( 5.61) |
| 5 | 70.000 | 150.571 | 0.610 | 19.429 | 14 ( 3.74) |
| 7 | 14.000 | 157.500 | 0.649 | 18.500 | 2 ( 0.53) |

Table 4.2: Distribution of Word Gain over the Devtest Set

| data set | SWB and CH | | | SWB only | | | CH only | | |
|---|---|---|---|---|---|---|---|---|---|
| | Utts. | true WER | opt. WER | Utts. | true WER | opt. WER | Utts. | true WER | opt. WER |
| train | 271 | 45.05 | 30.75 | 100 | 38.34 | 28.22 | 171 | 50.83 | 32.93 |
| test | 103 | 40.50 | 29.83 | 54 | 36.36 | 27.42 | 49 | 47.89 | 34.11 |
| Total | 374 | 43.51 | 30.41 | 154 | 37.52 | 27.84 | 220 | 50.08 | 33.17 |

Table 4.3: Characteristics of Train and Test Set (WER in %)

was randomly split into a training and a test set. Table 4.3 gives the major characteristics of these.

### 4.2.2 Data Subsets Used for Specific Evaluations

From the test data, 21 hypotheses (ca. 20%) were extracted which had a minimum word gain of three words. The rationale behind this set, called `eval21`, is to see how well the system performs in a case where it is working only on those Nbest lists which contain a fairly high potential of word gain.[3]

For a discriminative evaluation of the effects of (i) optimal tagging, and (ii) optimal segmenting (*in addition to optimal taging*), a subset of this `eval21` set was created, containing 19 of these 21 utterances, but only the (originally) first hypothesis, the true best and the true worst hypothesis were included. Additionally, the reference (i.e., transcript) of each utterance was added to this data set, s.t. in total we have four "hypotheses" in each of these 19 utterances. These datasets are called `only4, only4-opttag`, and `only4-optsegm` respectively.

[3]If we had fairly reliable confidence annotations (Chase, 1997), we could in fact approximate this biased selection. The reason is that there is a correlation between the average WER on the first best hypothesis and the potential word gain of an utterance (see Table 4.2).

### 4.2.3 Data Used for LM Construction, Grammar Development and POS Tagger Training

For training of the ngram model, for a basis of the development of the POS based grammar, and for the (re-)training of the POS tagger, we used the (standard) Switchboard corpus (Godfrey et al., 1992); no data from our Devtest set was used for these tasks. There were two reasons for that: (i) the robustness of our system should increase; (ii) for POS tagging and particularly for creating an ngram model, more data is needed than there is available in the Devtest set itself.[4]

## 4.3 POS Tagger

We are using (Brill, 1994)'s part of speech (POS) tagger as an important preprocessing component of our system. As later evaluations prove, the performance of this component is very crucial to the whole system's performance. In particular, the segmentation module and the POS based Phoenix chunk parser heavily rely on the correct output of the tagger; hence, if words are tagged incorrectly, these modules' performance drops as a direct consequence, and finally, also the global performance of the system.

Brill's tagger is publicly available (via ftp) and comes as a version which is trained on the Wall Street Journal and Brown corpora. However, in our case, where we are dealing with spontaneous speech, we had to face two problems: First, the tagset is not optimal, e.g., there are no tags for marking hesitations or disfluent input. Secondly, written language differs from conversational spoken language quite considerably, so the tagger would have to be retrained for that task.[5] In total, a corpus of approx. 18000 words, drawn from the Switchboard corpus, was manually tagged. The tagset we were using for that, was slightly modified from the aforementioned tagged (but erroneos) Switchboard subset, which in turn was derived from the Penn Treebank and Brown Corpus tagsets (see section 3.2.3 for more details about the tagset).

After several incremental steps of labeling and retraining, the following tag accuracies were achieved (average by crossvalidation over five disjoint data sets):

- for the Switchboard corpus: 91.2%
- for the DevTest data:[6] 88.3%

Since approx. 14 tags are exchangeable with others because of our tolerant grammar, the error drops if we subtract these tag-errors from the total number. The tagging accuriacies *with respect to the POS grammar* are as follows:

- for the Switchboard corpus: 92.8%

[4]The total size of the Devtest set is approx. 3900 words, whereas the Switchboard corpus contains approx. 3 million words.

[5]A sizeable handannotated portion of the Switchboard corpus was availabe to us, but since it contained many serious labeling errors, it only was used for creating the initial dictionary.

[6]Here, a small subset of approx. 2000 words was manually tagged, too.

| correct tag | assigned tag | occ. of error | evaluation and comments |
|---|---|---|---|
| CC | WRB | 20 | no problem (similar syntactic function) |
| PREP | TO | 18 | could be eliminated if we dispense with TO-tag |
| RB | UH | 16 | removes too much (i.e. overgenerates babbles) |
| UH | AFF | 13 | ok (e.g., "yeah" is babble or affirmative) |
| PREP | RP | 11 | potential problem in parsing |
| PRPA | PRP | 11 | detto |
| NN | DT | 10 | detto |
| PREP | UH | 10 | removes too much (see RB/UH) |
| VB | VBP | 9 | ok, grammar tolerates that |
| WDT | DT | 9 | problem for segm. module (will miss boundary) |
| ANA | WDT | 8 | detto (but here: wrong boundary inserted) |
| NN | JJ | 8 | probably ok most of time (if JJ is within NP) |
| UH | VB | 8 | ok (non detected babble) |
| WDT | ANA | 8 | see ANA/WDT |
| CC | CCC | 7 | problem for segmenter: will miss boundary |
| NNP | NNS | 6 | ok (grammar is tolerant wrt. that) |
| VBP | UH | 6 | too much removed (see RB/UH) |
| VBP | VB | 6 | ok (grammar tolerates that) |
| CCC | CC | 5 | segmentation problem (opp. to CC/CCC) |
| PRP | UH | 5 | see RB/UH |
| VB | NN | 5 | potential serious problem (missed verb) |

Table 4.4: Most frequent errors of the retrained POS tagger

- for the DevTest data: 90.6%

This performance increase of about 1.5-2.0% seemed to make it promising to just reduce the tag set s.t. these "ambiguities" would not have to be present in neither the tag set nor the grammar. However, the results of using a reduced version of the tag set (26 instead of 40 tags) showed a small *decrease* in performance, compared to the latter results (approx. 0.25% lower accuracy). It seems that having more tag information helps the tagger more than a reduced set could potentially simplify the rules. Thus, we decided to stick to the original, larger tag set for the final system design.

Table 4.4 shows the most frequent errors on a subset of the tagged data (ca. 20%, i.e., 3500 words). (For the explanation of the tag names, see section 3.2.3.)

## 4.4 Chunk Parser

The evaluation of the chunk parser's accuracy was done on subsets of the `only4`, `only4-opttag`, and `only4-optsegm` data sets. For each of these sets, the first

| test set | words total | missing | wrong | superfl. | error rate (%) |
|---|---|---|---|---|---|
| only4 | 372 | 33 | 13 | 1 | 12.6% |
| opttag | 377 | 7 | 0 | 0 | 1.9% |
| optsegm | 372 | 10 | 0 | 1 | 3.0% |

Table 4.5: Performance of the chunk parser on three different test sets

20 hypotheses (i.e. 5 utterances each, comprising the reference and three hypotheses, see above), were evaluated as follows: For each word appearing in the chunk parser's output (including the skipped words), it was determined, whether it belonged to the correct chunk, or whether it had to be classified into one of these three error categories:

- "missing": either not parsed or wrongfully incorporated in another chunk
- "wrong": belongs to the wrong type of chunk
- "superfluous": parsed as a chunk that should not be there (because it should be a part of another chunk)

The results of this evaluation are given in Table 4.5. We see that an optimally parsed input is indeed crucial for the accuracy of the parser: it increases from about 87.4% to up to 98.1%. The higher number of errors in the `only4-optsegm` set is intuitively unplausible but probably not significant. (Abney, 1996b) reports a comparable per word accuracy of his CASS2 chunk parser (92.1%).

## 4.5 Global Evaluation: Nbest Rescorer

First, we ran a series of 288 experiments on the `only4-*` data sets, varying LMs, penalty-factors, sort keys, hypothesis-length-cutoffs, and length normalization parameters. The rescorer was set to work in "UNIX sort mode" (see section 3.2.7). The expected WER gain for these data sets is +0.8% (in the case of cutoff=0).

We got the following results:

- best performance: +7.7% WER gain in `only4-opttag`
- `only4-optsegm` is not better
- for `only4`, the best configuration yields +5.1% WER gain
- thus, correct tagging gives an improvement of more than 2.5% in WER gain
- using a low cutoff (cutoff=4) is ok

| data set | eval21 parameters | only4 parameters | expected WER gain |
|---|---|---|---|
| train | -2.2 | -3.5 | -5.6 |
| test | -2.7 | -4.3 | -4.9 |

Table 4.6: WER gain comparison of two parameter settings for the train and test set (in %)

- the reference hypotheses are sometimes *not* higher ranked than the true best hypotheses[7]
- sorting by chunk LM is better than by the chunk coverage score
- lm-penalty-factor 1.0 seems a good choice
- 3gram seems (slightly) better than 5gram

The next set of experiments was done in the same setting, but now with the (biased) `eval21` data set. The expected WER is +0.5%.

The results can be summarized as follows:

- best configuration yields +0.7% WER gain which is only very marginally better than the expected WER gain
- the trigram is (again) better than the fivegram
- a high cutoff is better
- chunk coverage score works better than the chunk LM score

We see that some of the trends from the `only4-*` data sets are reversed here; but since this `eval21` is a more realistic data set, we expected to get better results on the large test and train data sets when using these winning parameters rather than those from the `only4-*` data sets. As Table 4.6 shows, these assumptions were justified: the `eval21` settings outperform the `only4-*` settings by about 1.5% on both train and test set.[8] As a consequence, we kept these parameter settings for the next stage of experiments, where the rescorer was using a neural net as rescoring method.

The next stage of the evaluation involved the use of the neural net (NN) option in the rescorer (instead "UNIX sort"). In total, 80 different NNs were created, using the data in the train set:[9] we used 5 different numbers of learning

[7] E.g., in the winning configuration of `only4-opttag`, we have 12 winning, 3 losing, and 4 neutral utterances, but only in 5 cases, the reference hypothesis was actually ranked first.

[8] A possible explanation for the fact that we have a higher win over the expected WER in the train set as opposed to the test set may be that the train set contains a higher WER gain potential (14.4% as opposed to 10.7% for the test set).

[9] The range of the NN parameters were determined partly by very early experiments in our system development phase.

| data set | best performance | expected WER gain |
|---|---|---|
| eval21 | +2.0 | +0.5 |
| **test** | **+0.3** | **-4.9** |

Table 4.7: WER gain: best results in NN experiments for two test sets (in %)

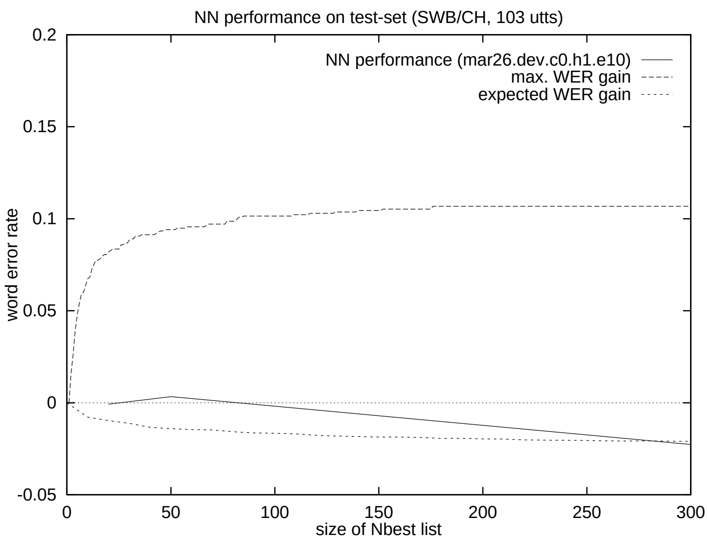


Figure 4.2: NN performance on the test set (nbest_cutoff=20, 50, 300)

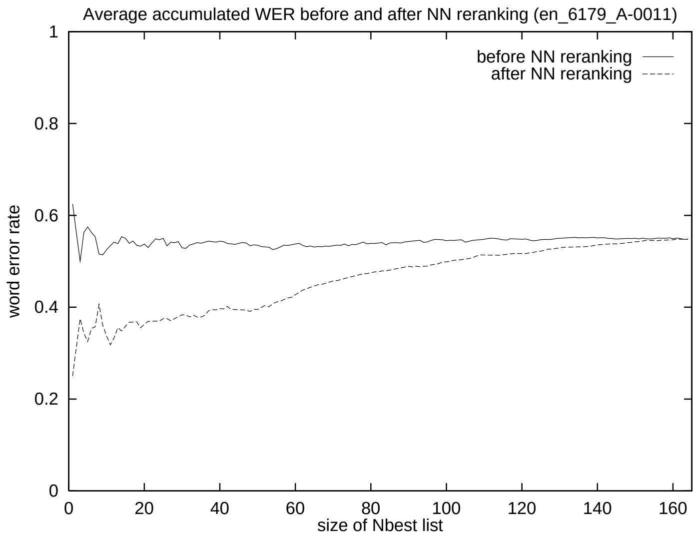


Figure 4.3: Cumulative average WER before and after reranking for an example utterance

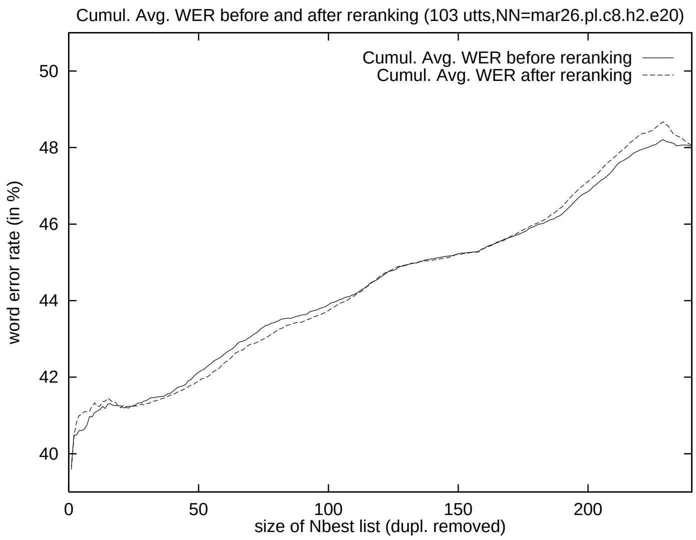


Figure 4.4: Cumulative average WER before and after reranking of the test set

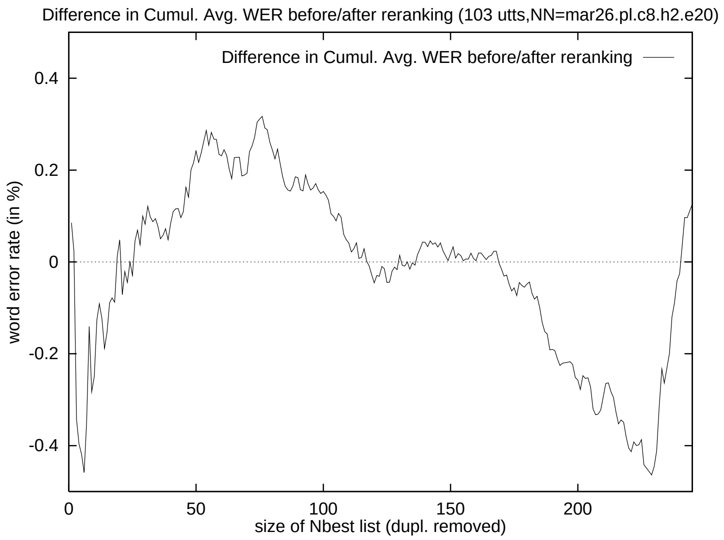


Figure 4.5: Difference in cumulative average WER for the test set (before and after reranking)

epochs (5, 10, 20, 30, 50), 2 different numbers of hidden units (1, 2), the two different types of chunk LM scores, and 4 different cutoff factors (0, 4, 8, 12). We now tested the data from the test set with these nets; for each test, both the NN with the *same* cutoff and with *no* cutoff was used. The idea behind this is that while on the one hand a larger training set may be helpful, on the other hand a training set that is more similar in its data to the test set may be better, even if it would be smaller. Also, we used three different so-called *nbest-cutoffs* for each experiment. Unlike the (standard) cutoffs, which just eliminate short utterances from the reranking procedure, these nbest-cutoff disallow hypotheses with a number *higher* than this nbest-cutoff to move to the first position. This amounts to working with a reduced size of the Nbest list. We chose the parameter `nbest_cutoff`=20, 50, and 300. (The last value corresponds to using the full Nbest-list). The idea behind this parameter is to limit the potential losses in WER gain by "distractors" from far behind in the Nbest list.

Since we had 80 NNs, 2 modes, and 3 nbest-cutoff factors, the total number of experiments was 480 for the `test` and `eval21` sets. Table 4.7 gives the best results of these experiments, while Figure 4.2 shows the performance of this net at three distinct nbest_cutoff data points (20, 50, and 300), compared to the maximum and the expected WER gain.[10]

[10]The best NN for the test set was `mar26.dev.c0.h1.e10`, i.e., using the whole training data (c0), having one hidden unit (h1), and training for 10 epochs (e10). A cutoff of 8 was used; the nbest_cutoff was 50.

From this we see that using NNs is not only better than just using a UNIX sort approach but that we even manage to get a small, but *positive* result in WER on a non-biased test set. While this effect is quite small, one has to keep in mind that the (constituent-like) chunk representations were the *only* source of information for our reranking system. It can be expected, that including more sources of knowledge, like the plausibility of correct verb-argument structures (the correct match of subcategorization frames) and the likelihood of selectional relations between the verbal heads and their head noun arguments would further improve these results. Another factor hich might have a significant influence is the WER improvement achieved meanwhile by the speech recognizer itself: on SWB data, the WER dropped from about 37.5% in the data we were using to about 26% recently.[11] Even if the overall properties of the data don't change too much with this improvement, it is very likely that the potential WER gain drops proportionally to this speech recognizer improvement, and that, as a consequence, hypotheses with high WER gain are becoming more sparse and harder to identify as such.

Departing from our main evaluation criterion, the WER gain on the first best hypothesis, we calculated the cumulative WER *before* and *after* reranking, over the size of the Nbest list for various hypotheses. Figure 4.3 shows the plots of these two graphs for an example utterance. We see very clearly, that in this example not only the new first hypothesis has a WER gain compared to the old one, but that *in general* hypotheses with lower WER moved towards the top of the Nbest list.

Looking at the whole test set, this trend is obviously much weaker than in this specific example, but it is still there, as Figure 4.4 shows. Finally, Figure 4.5 plots just the difference of the graphs of Figure 4.4. While we see a small negative effect of reranking very close to the first element in the Nbest list, there is a markably positive effect in the upper half of the list, compensated by a "deficit" towards the end, where the hypotheses with higher WER moved to. These general trends are also very encouraging and show that the NN reranking component behaves in the right and desired way.

## 4.6 Human Study

In order to assess the ability of **humans** to distinguish speech recognizer hypotheses in terms of "structural well-formedness" (syntax) and "meaningfulness" (semantics), a small study was undertaken.

### 4.6.1 Data

From the train set, from all utterances which could in theory be improved, the first best and the true best hypothesis are extracted and then scrambled, both in utterance order and in first/true-best order. (True best: lowest WER).

[11] Michael Finke, personal communication

The data is sent through our preprocessor modules; the POS tags are removed and slashes are inserted instead of eos/EOS marks.

To minimize errors of the babble-filter, the following babbles were **not** extracted: *like, then, i guess, kind of.*

Finally, all short utterances (at least one hypothesis fewer than 5 tokens) were removed since they are also not too interesting for rescoring either.

The final set consists of 128 hypothesis-pairs.

### 4.6.2 Task

Human subjects should read each hypothesis-pair and then decide, which of the two hypotheses is "better" from (a) a structural and (b) a meaning point of view. If a decision can't be made, they should fill in a question mark.

### 4.6.3 Evaluation

#### 4.6.3.1 Judgement Classification

There are 9 possibilities for the comparison of each hypothesis-judgement (syntactic/semantic) between two subjects:

A A, A B, A ?, ? A, ? B, ? ?, B A, B B, B ?

These are grouped in the following way:

1. strong agreement: A A, B B (i.e. same judgement from both subjects)
2. weak agreement: A ?, ? A, ? B, B ? (one subject did no judgement)
3. strong disagreement: A B, B A (i.e. different judgements)
4. undecided: ? ? (hypothesis was not judged by both subjects)

With respect to the potential "gain", when these judgements were to be used for the purpose of reranking, the following scenarios were distinguished (intra-subject evaluation; the first letter corresponds to the syntactic, the second letter to the semantic judgement of one subject for one hypothesis; in brackets: the hypothesis with the lower WER).

1. uniform win: A A (A), B B (B)
2. syntactic win: A ? (A), B ? (B) (no semantic judgement)
3. semantic win: ? A (A), ? B (B) (no syntactic judgement)
4. syntax wins over semantics: A B (A), B A (B)
5. semantics wins over syntax: A B (B), B A (A)
6. wrong choice: A A (B), B B (A), A ? (B), ? A (B), B ? (A), ? B (A)
7. no choice: ? ?

| Subject | Syntax pref. | Sem. pref. |
|---|---|---|
| 0 | 10.0 | 10.3 |
| 1 | 10.0 | 10.2 |
| 2 | 9.1 | 9.7 |
| 3 | 10.2 | 10.8 |
| Total Avg. | 9.8 | 10.2 |

Table 4.8: Human Performance (WER gain in %)

#### 4.6.3.2 Improvement in WER if judgement was used for reranking

This was calculated in two ways:

1. uniform_win + syntactic_win + semantic_win + syn_over_sem + no_choice/2
2. uniform_win + syntactic_win + semantic_win + sem_over_syn + no_choice/2

The first scenario assumes superiority of syntactic judgements in case of a conflict, the second assumes the converse. The "no_choice/2" was added simply to emulate a random process of picking either hypothesis in case of the lack of any judgements.

While the maximum WER gain for these 128 hypothesis-pairs is 15.2%, the expected WER gain (i.e., the WER gain of a random process) is 7.6%.

Table 4.8 shows the WER gain when these two scores would be used and the human judgements were to be used for reranking.

Whereas the difference between both methods to a random choice is highly significant (syntax: $\alpha = 0.01, t = 9.036, df = 3$; semantics: $\alpha = 0.01, t = 11.753, df = 3$)[12] , the difference between these two methods is **not** ($\alpha = 0.05, t = -1.273, df = 6$)[13]. The latter is most likely due to the fact that there were only few hypotheses that were judged *differently* in terms of syntactic or semantic well-formedness by one subject: on average, only 6% of the hypothesis-pairs received a different judgement by one subject (`sem_over_syn` or `syn_over_sem`).

**Note:** A NN trained on the test-set[14] was able to get a significantly better result than these subjects: 12.8% (syntax: $\alpha = 0.01, t = -12.082, df = 3$; semantics: $\alpha = 0.01, t = -11.310, df = 3$)[15].

#### 4.6.3.3 Inter-subject agreement on the judgements

Table 4.9 gives the percentage of agreement between subjects in the four classes of agreement (syntactic/semantic) listed above. We see that humans strongly

[12]These results were obtained using the one-sided t-test.
[13]Two-sided t-test.
[14]We took the best net from our system evaluation for that purpose.
[15]One-sided t-test.

| Subj. A vs. Subj. B | strong agr. (2/9) | weak agr. (4/9) | strong disagr. (2/9) | don't know (1/9) |
|---|---|---|---|---|
| 0 vs. 1 SYN | 39.06 | 46.09 | 9.38 | 5.47 |
| 0 vs. 1 SEM | 50.78 | 28.13 | 11.72 | 9.38 |
| 0 vs. 2 SYN | 33.59 | 44.53 | 17.19 | 4.69 |
| 0 vs. 2 SEM | 54.69 | 29.69 | 12.50 | 3.12 |
| 0 vs. 3 SYN | 35.94 | 33.59 | 10.16 | 20.31 |
| 0 vs. 3 SEM | 37.50 | 25.00 | 15.63 | 21.88 |
| 1 vs. 2 SYN | 56.25 | 12.50 | 30.47 | 0.78 |
| 1 vs. 2 SEM | 61.72 | 17.19 | 20.31 | 0.78 |
| 1 vs. 3 SYN | 50.78 | 25.00 | 19.53 | 4.69 |
| 1 vs. 3 SEM | 42.97 | 32.81 | 14.84 | 9.38 |
| 2 vs. 3 SYN | 50.78 | 25.00 | 21.09 | 3.12 |
| 2 vs. 3 SEM | 43.75 | 34.38 | 18.75 | 3.12 |
| TOTAL SYN | 44.40 | 31.12 | 17.97 | 6.51 |
| TOTAL SEM | 48.57 | 27.86 | 15.63 | 7.94 |

Table 4.9: Inter-Subject Agreement (in %)

agree on average in almost 50% of the cases[16], whereas they disagree only in about 15% of the cases.[17] Also, one can observe that semantic agreement is slightly higher than syntactic agreement.

[16] random expectation: 22%

[17] again, random expectation: 22%

# Chapter 5

# Future Work

## 5.1 Further improvement of the POS tagger

Since the POS tagger proved itself to be of central importance for the performance of our system, it might well be worthwhile in investing some amount of effort to improve its accuracy. One (very time-consuming) way to do that would be to make experiments on different tag sets, another possibility involves experiments with "cleaning-up"-preprocessors (similar to the pipe we are using), s.t. already the *input* of the POS tagger is of a more "regular" form.

Finally, it might be just necessary to extend the amount of carefully hand-labelled data to provide the training procedures with more examples to learn from.

## 5.2 Alternative Language Models

So far, all our LMs were based on simplex clauses; it could be the case that when we move to the hypothesis level, we would get a better (because longer ranging) model and therefore better scores.

Another idea for improvement is to integrate skipped words (or sections) into the LM (similar to the modeling of noise in speech) and that way to get rid of the skipping penalties we were using so far and which blurred the statistical nature of the model. (Maybe, we could totally dispense then with the chunk coverage scores, since these would be modeled already by such a chunk LM comprehensively.)

## 5.3 Using scores from other parsers

It is not a priori clear, why one might not prefer a *combination* of scores from various parsers to the score of just the single chunk parser. A feasable and obtainable option would be, e.g., to adapt the Link Grammar parser (Grinberg et al., 1995) to our task and use the linkage scores of the best parse for every hypothesis (or simplex clause) as an additional (syntactic) "wellformedness-indicator".

## 5.4 Identifying Good Reranking Candidates

So far, the only and exclusive heuristics we are using for determining when to rerank and when not to, is to use the length-cutoff parameter to exclude (too) short utterances from being considered in the final reranking procedure.

As Lin Chase has shown in her thesis (Chase, 1997), there are a number of potentially useful "features" from various sources within the recognizer which can predict, at least to a certain extent, the "confidence" that the recognizer has about a particular hypothesis. We saw in table 4.2 that hypotheses which have a higher WER on average also exhibit a higher word gain potential. So it is not unreasonable to believe that a combination of some of these scores can actually help in finding out a better set of reranking candidates than just working with a cutoff filter. Particularly, if we recall that in our `eval21` set we were able to get a WER gain of about +2%, we think that these lines of investigations deserve some more thoughts and experiments.

## 5.5 Improving the Neural Nets

The neural nets are currently trained to predict the `absolute` WER of a hypthesis. While this may be very useful for an application we talked about in the previuos section, we think that for the rescoring task itself other kinds of target values might make more sense, e.g., using the *relative* WER difference wrt. the true best WER in the Nbest list.

## 5.6 Adding Argument Structure Representations

We already mentioned earlier that in linguistic terms the chunk representation our system currently exploits is a very weak one: it only gives an idea about which constituents there are in a clause and how their ordering is. A richer model has to include also the *dependencies* between these chunks.

The first step would be to produce statistics about verb-argument structures from a large corpus and use these probabilities in a "mapper" module that tries to find an optimal match between the chunks found in a clause and the possible subcategorization frames for the main verbal head.

Finally, one could compute statistics about selectional restrictions from these verbal heads wrt. their arguments. To avoid the sparse data problem, a sensible clustering of the nouns has to be done, e.g., along the work in unsupervised word sense disambiguation (Schütze, 1992).

# Chapter 6

# Summary and Conclusions

With the work in this project I have shown that it is feasible to produce chunk based representations for spontaneous speech in unrestricted domains on a high level of accuracy. First, a rule based POS tagger attaches tags to every word in the input. Its performance proves crucial for subsequent modules of the system. Next, a preprocessing pipe performs the task of "cleaning up" the input from a lot of spontaneous speech phenomena that otherwise would put a substantial burden on the task of later components, such as parsers and semantic mappers. Then, the input is segmented into smaller units ("simplex clauses") which are parsed by the POS based Phoenix chunk parser.

The chunk representations generated by this parser are then used to generate scores for an Nbest list reranking component.

The results are promising, in that the best performance on a randomly selected test set is a decrease in word error rate of 0.3 percent, mesaured on the new first hypotheses in the reranked Nbest lists. Although this gain is quite small, one has to take into account that more linguistic information could be used for further enhancements (e.g., a combination of subcategorization frames and selectional restrictions of verbal heads).

# Bibliography


Steven Abney. 1990. Rapid incremental parsing with repair. In *Proceedings of the 6th New OED Conference: Electronic Text Research*, pages 1–9.

Steven Abney. 1994. Partial parsing. Tutorial at ANLP-94 (slide series).

Steven Abney. 1996a. Chunk stylebook. MS ("work in progress", see http://sfs.nphil.uni-tuebingen.de/˜abney/Papers.html#96i).

Steven Abney. 1996b. Partial parsing via finite-state cascades. MS ("work in progress", see http://sfs.nphil.uni-tuebingen.de/˜abney/).

Eric Brill. 1994. Some advances in transformation-based part of speech tagging. In *Proceeedings of AAAI-94*.

Eric Brill. 1995. Unsupervised learning of disambiguation rules for part of speech tagging. In *Proceedings of the 3rd Workshop on Very Large Corpora*.

Eugene Charniak. 1993. *Statistical Language Learning*. MIT Press, Cambridge, MA.

Lin Chase. 1997. *Error-responsive feedback mechanisms for speech recognizers*. Ph.D. thesis, Carnegie Mellon University, Pittsburgh, PA.

Noam Chomsky. 1965. *Aspects of the Theory of Syntax*. MIT Press, Cambridge, MA.

Marsal Gavaldà, Klaus Zechner, and Gregory Aist. 1997. High performance segmentation of spontaneous speech using part of speech and trigger word information. In *Proceedings of the 5th ANLP Conference, Washington DC*, pages 12–15.

J. J. Godfrey, E. C. Holliman, and J. McDaniel. 1992. SWITCHBOARD: telephone speech corpus for research and development. In *Proceedings of the ICASSP-92*, volume 1, pages 517–520.

Dennis Grinberg, John Lafferty, and Daniel Sleator. 1995. A robust parsing algorithm for link grammars. Technical report, School of Computer Science, CMU, CMU-CS-95-125, August.

Ralph Grishman and Beth Sundheim. 1996. Message understanding conference 6 – a brief history. In *Proceedings of COLING-96*, pages 466–471.

Ralph Grishman. 1995. The NYU system for MUC-6 or: Where's the syntax? In *Sixth Message Understanding Conference (MUC-6)*, pages 167–175.

Jerry R. Hobbs, Douglas E. Appelt, John S. Bear, David J. Israel, and W. Mabry Tyson. 1992. FASTUS: A system for extracting information from natural language text. Technical report, SRI International, Menly Park, CA, November.

Jerry R. Hobbs, Douglas Appelt, John Bear, David Israel, Megumi Kameyama, Mark Stickel, and Mabry Tyson. 1996. FASTUS: A cascaded finite-state transducer for extracting information from natural language text. WWW document, SRI International.

Rosie Jones. 1996. Re-ranking speech hypotheses using syntactic information from the Link Grammar parser. Project Report, Computational Linguistics Program, CMU.

Walter Kasper and Hans-Ulrich Krieger. 1996. Modularizing codescriptive grammars for efficient parsing. In *Proceedings of COLING-96*, pages 628–633.

Walter Kasper, Hans-Ulrich Krieger, and Jörg Spilker. 1996. From word hypotheses to logical form: An efficient interleaved approach. Technical report, Verbmobil Report.

Martin Kay, Jean Mark Gawron, and Peter Norvig. 1994. *Verbmobil. A Translation System for Face-to-Face Dialog.* Chicago University Press, Chicago.

Kimmo Koskenniemi. 1990. Finite state parsing and disambiguation. In *Proceeedings of COLING-90*, pages 229–232.

K. Lari and S. J. Young. 1990. The estimation of stochastic context-free grammars using the inside-outside algorithm. *Computer Speech and Language*, 4:35–56.

Alon Lavie. 1996. *GLR*: A Robust Grammar-Focused Parser for Spontaneously Spoken Language.* Ph.D. thesis, Carnegie Mellon University, Pittsburgh, PA.

Marc Light. 1996. CHUMP: Partial parsing and underspecified representations. In *Proceedings of the 12th European Conference on Artificial Intelligence (ECAI-96), Budapest, Hungary.*

David M. Magerman and Mitchell M. Marcus. 1991. PEARL: A probabilistic chart parser. WWW document.

Fernando C.N. Pereira and Rebecca Wright. 1996. Finite-state approximation of phrase-structure grammars. WWW Computational Linguistics Archives.

Carl Pollard and Ivan A. Sag. 1994. *Head-Driven Phrase Structure Grammar.* Uiniversity of Chicago Press, Chicago & London.

Yan Qu, Carolyn P. Rosé, and Barbara Di Eugenio. 1996. Using discourse predictions for ambiguity resolution. In *Proceedings of COLING-96*, pages 358–363.

Klaus Ries, Bernhard Suhm, and Petra Geutner. 1997. Language modeling in janus. file://localhost/afs/cs/project/cmt-38/ries/lmdok/janus-lm.doku/janus-lm.doku.html.

Hinrich Schütze. 1992. Dimensions of meaning. In *Proceedings of Supercomputing 92.*

Jeff Shufelt. 1994. Backpropagation neural net software package. Carnegie Mellon University.

Daniel D. K. Sleator and Davy Temperley. 1991. Parsing English with a Link Grammar. Technical report, Carnegie Mellon University, Pittsburgh, PA, October.

Bernhard Suhm, Brad Myers, and Alex Waibel. 1996. Interactive recovery from speech recogniztion errors in speech user interfaces. In *Proceedings of the ICSLP-96*, pages 865–868.

Masaru Tomita and Eric H. Nyberg. 1988. Generation kit and transformation kit (version 3.2). CMU-CMT-88-MEMO.

Masaru Tomita. 1990. The generalized LR parser/compiler - version 8.4. In *Proceedings of COLING-90*, pages 59–63.

Marc Vilain and David Day. 1996. Finite-state phrase parsing by rule sequences. In *Proceedings of COLING-96*, pages 274–279.

Marc Vilain and David Palmer. 1996. Transformation-based bracketing: Fast algorithms and experimental results. In *Workshop on Robust Parsing, 8th European Summer School in Logic, Language and Information, Prague, Czech Republic*, pages 93–102.

Wolfgang Wahlster. 1993. Verbmobil — translation of face-to-face dialogs. In *Proceedings of MT Summit IV, Kobe, Japan.*

Alex Waibel, Michael Finke, Donna Gates, Marsal Gavalda, Thomas Kemp, Alon Lavie, Lori Levin, Martin Maier, Laura Mayfield, Arthur McNair, Ivica Rogina, Kaori Shima, Tilo Sloboda, Monika Woszczyna, Torsten Zeppenfeld, and Puming Zhan. 1996. JANUS-II - advances in speech recognition. In *Proceedings of the ICASSP-96.*

Wayne Ward. 1991. Understanding spontaneous speech: The PHOENIX system. In *Proceedings of ICASSP-91*, pages 365–367.

Wayne Ward. 1994. Extracting information in spontaneous speech. In *Proceedings of the ICSLP 94, Yokohama, Japan*, pages 83–86.

Klaus Zechner. 1997. A literature survey on information extraction and text summarization. MS, available at http://www.contrib.andrew.cmu.edu/ žechner/klaus.html.

Puming Zhan, Klaus Ries, Marsal Gavaldà, Donna Gates, Alon Lavie, and Alex Waibel. 1996. JANUS-II: Towards spontaneous spanish speech recognition. In *Proceedings of the ICSLP-96*, pages 2285–2288.

# Appendix A

# Sample Files

## A.1 Excerpt from an Nbest-list

Explanation:

Rows with a # are comments. They contain information about the database, dialogue number, speaker, turn number, start frame (time), global speech recognizer score, and hypothesis rank (in that order).

The other rows represent word hypotheses; their content is as follows:

| Column Nr | Description |
|---|---|
| 1 | utterance number |
| 2 | speaker ID |
| 3 | start frame |
| 4 | duration |
| 5 | word hypothesis |

```
# en_4792_A-0000 7.73 2581.57 1
en_4792 A    7.73    0.04 (
en_4792 A    7.77    0.12 YOU
en_4792 A    7.89    0.24 KNOW
en_4792 A    8.13    0.01 )
# en_4792_A-0000 7.73 2602.21 2
en_4792 A    7.73    0.04 (
en_4792 A    7.77    0.15 YOU
en_4792 A    7.92    0.21 ALL
en_4792 A    8.13    0.01 )
# en_4792_A-0000 7.73 2604.96 3
en_4792 A    7.73    0.04 (
en_4792 A    7.77    0.12 YOU
en_4792 A    7.89    0.24 DON'T
en_4792 A    8.13    0.01 )
```

## A.2 Hypothesis List File

```
you weren't born just to soak ups on
you weren't born just is so cups on
you weren't boring just it's so cups on
you weren't born justice so groups on
you weren't born justice so crops on
you weren't born just just so groups on
you weren't board justice so cups on
you weren't born just just so crops on
```

## A.3 General Utterance Log Information File

Contents of this file:

| Column Nr | Description |
|---|---|
| 1 | utterance label |
| 2 | total number of hypotheses in the Nbest list |
| 3 | total number of *non-duplicates* in the Nbest list |
| 4 | average reference length |

```
en_4792_A-0001 300 190 32.5366666666667
en_4792_A-0005 300 81 12.8966666666667
en_4792_A-0007 16 7 1.8125
en_4792_B-0001 300 133 7.79333333333333
en_4792_B-0002 300 159 17.4233333333333
en_4801_A-0006 300 136 34.73
en_4829_A-0006 300 178 7.15333333333333
en_4829_B-0001 300 60 8.37
en_4829_B-0002 300 105 9.07333333333333
en_4829_B-0009 300 128 12.2033333333333
```

## A.4 Individual Hypothesis Log Information File

Contents of this file:

| Column Nr | Description |
|---|---|
| 1 | utterance label |
| 2 | hypothesis number in the reduced Nbest list (no duplicates) |
| 3 | hypothesis number in the original Nbest list |
| 4 | normalized speech recognizer score |
| 5 | length of reference |
| 6 | word error rate |

```
en_4792_A-0000 1 1 1 1 2
en_4792_A-0000 2 2 0.996666666666667 1 2
en_4792_A-0000 3 3 0.993333333333333 1 2
en_4792_A-0000 4 4 0.99 1 2
en_4792_A-0000 5 11 0.966666666666667 1 2
en_4792_A-0000 6 21 0.933333333333333 1 2
en_4792_A-0000 7 24 0.923333333333333 1 3
en_4792_A-0000 8 26 0.916666666666667 1 3
en_4792_A-0000 9 27 0.913333333333333 1 3
```

## A.5 Simplex Clause Segmentation Index File

Contents of this file:

| Column Nr | Description |
|---|---|
| 1 | start-line of the next hypothesis in the list of simplex clauses |

```
0
7
14
21
28
36
43
50
57
64
```

## A.6 Phoenix forms File

Explanation: Each chunk can appear as a top level slot in a chunk sequence (called `chunks`); there are 20 different chunks, including the `eos` chunk which marks the boundary between two simplex clauses.

```
FUNCTION: chunks
NETS:
[_np]     # standard NP, also numerals (e.g [the four/CC] of us)
[_pp]     # standard PP: prep+NP (but only acc. PRPA as arg,)
[_vb]     # verb-group: all verbal stuff + interspersed adv's
[_vbneg]  #  negated verb-group
[_adv]    # adverbial phrase
```

```
[_conj]   # all conjunctions (CC, CCC, also as-far-as etc.)
[_wh]     # an oblique wh-phrase (e.g. what do you prefer?)
[_adjp]   # adj.-phrase
[_toinf]  # inf's and gerunds, introduced by prep./"to"
[_vpart]  # split verbal particle
[_aux]    # auxiliary phrase
[_auxneg] # negated auxiliary phrase
[_rpro]   # relative pronoun
[_whnp]   # wh-np: e.g. [which movie] do you like?
[_expl]   # expletive: there are...
[_comm]   # comment: that's ...
[_neg]    # negation-element (no, I...)
[_aff]    # affirm. element  (yes, ...)
[_misc]   # currently just CV tag (bye/CV)
[_eos]    #  end of sx-clause marker
;
```

## A.7 Phoenix nets File

Explanation: Each chunk is a `net`, additionally, the head-words are also `nets`.

```
_np
_pp
_vb
_vbneg
_aux
_auxneg
_misc
_adv
_conj
_comm
_expl
_wh
_adjp
_toinf
_neg
_aff
_vpart
_rpro
_eos
_whnp
_p_head
```

```
_n_head
_vb_head
```

## A.8 Phoenix MAP_STRINGS File

Explantion: For our application, this file was not employed. However, it would allow, e.g., to use a tagger with a different tag set and to map it onto the tag set we used in our grammar. (Each line contains a pair of strings that specify the old and the new name.)

```
‘‘dummy-old’’, ‘‘dummy-new’’
```

## A.9 POS Grammar Rule Files

Explanation: This is a concatenation of grammar and non-terminal files. Each `net` is represented in a single file, starting with the net's name in square brackets and followed by the right hand side rule expansions, enclosed in round brackets. Symbols in lower case denote terminals (here: part of speech tags), symbols in upper case denote non-terminals that are further expanded either in the same file or in an include-file (non-terminal file), shared by various grammar files.

```
[_adjp] # e.g. i am interested, i am [(very) busy]...
(JJ_MOD)

#include JJ.nt

[_adv] # adverbial phrase
(+RB)
(*prep prep +RB) # around here, at first, at most, out of nowhere...

RB
(rb)
(rbr)
(rbs)
[_aff]
(aff)
[_aux]  # aux. phrase: in questions, e.g. [do] you see...
(aux)

[_auxneg]
(aux-n)

[_comm] # comments: "that's..."
```

```
(dt-aux)

[_conj]
(ccc)  # constit.-conj (NEW tag, by hand)
(*cc cc)   # sentence-conj; e.g. even if...
# (cc jj cc) # as soon as... now handled in multi-words.pl
# (cs rb cc) # as well as... ...ditto...

[_eos]
(eos)

[_expl] # expletives ("there's"...)
(ex-aux)
(ex)

[_misc]
(cv) # conventional form (bye/CV)

[_n_head]
(NN)

#include NN.nt

[_neg]
(neg)

[_np]
(*dt *dt   +*[_adjp] +[_n_head])
                          # *+JJ_MOD instead of *JJ_MOD (all the
                          #                 four brave kids....)
(*dt *prp$ +*[_adjp] +[_n_head]) # "all my four brave kids"
(prp)                   # you/I
(prpa)                  # you/me
(*dt *dt cd)   # 21.2. all the four (of us)
                       # also: "[channel] [eleven]"=> _np _np
(*dt ana) # (all) this/that...

#include JJ.nt
#include NN.nt

[_p_head]
(+prep) # multiple: out of..., despite of ...

[_pp]
([_p_head] [_np])
([_p_head] prpa) # accusative pronoun
```

```
[_rpro] # relative clause intro
(wdt)

[_toinf]
(to  *rb [_vb_head])  # e.g. want [to come]
(prep *rb [_vb_head]) # gerund: [for (always) doing]....

[_vb]
(*VBAUX *VBAUX *rb *+VB [_vb_head])
#  e.g. "would have strictly opposed the decision"...

# !!! changes here should be mirrored in _vbneg.gra !!!

#include VB.nt
[_vb_head]
(vbz)
(vbp)
(vbd)
(vbg)
(vbn)
(vb)

[_vbneg]
(aux-n *VBAUX *rb *+VB [_vb_head])
(+VBAUX neg *VBAUX *rb *+VB [_vb_head])

#include VB.nt
[_vpart] # stand-alone verb-particle, e.g. took... off
(rp)

[_wh]   # oblique wh-phrase (e.g. when do you come? how well does he perform?)
(wrb *jj)
(wrb *rb)

[_whnp] # noun phrase with wh-mod: which train did you take?
(wp *+[_adjp] +[_n_head])

#include JJ.nt

JJ_MOD   # modified adjective
(*rb JJ_ONLY)
(*rb vbg)     # e.g. a flying saucer
                      # prob. exclude in a strict version
JJ_ONLY
(jj)
```

```
(jjs)
(jjr)
(cd)
NN
(nn)
(nns)
(nnp)
(nnps)

VB
(vbz)
(vbp)
(vbd)
(vbg)
(vbn)
(vb)

VBAUX
(vbz)
(vbp)
(vbd)
(vbg) # ideally: no gerund
(vbn) # ideally: no past perf.
(vb)  # ideally: no inf.
(aux)
```

## A.10 Parser Output Sample

Explanation: This is the standard Phoenix parser output, given the parameter settings we are using in our system (see section 3.3.5). The relevant lines for our system are:

- lines starting with `;;;` — they give the current line number of the input file, i.e., the number of the simplex clause
- lines starting with > — they give a summary over the parse results: all tokens that could not be parsed, are prefixed with a `%`
- adjacent lines starting with a square bracket *after* the line with a `;` — these give the actual parses of the chunks

```
sizeof Edge= 176
sizeof Edge Buffer= 8800000
sizeof Intp= 6504
sizeof IntpBuffer= 1626000
```

```
There are 1 forms
***********************************

;;;1
 EOS
#1
 parse time 0  milliseconds
edges used= 1      max= 50000
interpretations produced= 1  max intps used= 1    max= 250
string 1  parse 1
> eos
Interpretation score 1
Frame chunks    score= 1    num_slots= 1
[_eos] ( EOS )


;
Frame chunks    score= 1    num_slots= 1
[_eos] ( EOS )


;;;2
 PRP DT
#2
 parse time 0  milliseconds
edges used= 2      max= 50000
interpretations produced= 1  max intps used= 1    max= 250
string 2  parse 1
> prp %DT
Interpretation score 1
Frame chunks    score= 1    num_slots= 1
[_np] ( [_pn_head] ( PRP ))


;
Frame chunks    score= 1    num_slots= 1
[_np] ( [_pn_head] ( PRP ))
```

## A.11 POS Filter Output File

Explanation: This is an extraction from the Phoenix parser output file. Just the lines starting with a > are extracted.

```
> eos
> prp %DT
> prp aux-n
> prp rp
> prp rb
> prp
```

## A.12 Chunk Filter Output File

Explanation: This is an extraction from the Phoenix parser output file. Just the adjacent lines starting with a square bracket after a line with a single ; are extracted. Each `delim` marks the beginning of the next simplex clause.

```
delim
 [ _eos ]    (   EOS   )
delim
 [ _np ]    (    [ _pn_head ]    (   PRP   )   )
delim
 [ _np ]    (    [ _pn_head ]    (   PRP   )   )
 [ _auxneg ]    (   AUX-N   )
delim
 [ _np ]    (    [ _pn_head ]    (   PRP   )   )
 [ _vpart ]    (   RP   )
delim
 [ _np ]    (    [ _pn_head ]    (   PRP   )   )
 [ _adv ]    (   RB   )
delim
 [ _np ]    (    [ _pn_head ]    (   PRP   )   )
```

## A.13 Chunk Sequence File

Explanation: Each line contains the sequence of chunks found by the chunk parser for a simplex clause.

```
np
np auxneg
np vpart
np adv
```

np

## A.14 Score Matrix File

Contents of this file:

| Column Nr | Description |
|---|---|
| 1 | WER for training, dummy (e.g., 0.0) in `eval`-mode |
| 2 | normalized speech recognizer score |
| 3 | chunk language model score |
| 4 | chunk coverage score |
| 5 | skipped words score |
| 6 | skipped sections score |

```
0 1 5.59443 1 0 0
0 0.996666666666667 9.04159 0.5 0.5 0.5
0 0.993333333333333 5.24435 1 0 0
0 0.99 5.24435 1 0 0
0 0.966666666666667 5.24435 1 0 0
0 0.943333333333333 8.04159 1 0 0
```

## A.15 First Best Extraction Information File

Contents of this file:

| Column Nr | Description |
|---|---|
| 1 | utterance label |
| 2 | hypothesis number to be extracted for the new first-best list |

```
en_4792_A-0001 212
en_4801_A-0006 180
en_4829_A-0006 246
en_4829_B-0009 66
en_5872_A-0006 49
en_6047_B-0012 46
en_6071_B-0003 174
en_6071_B-0009 276
en_6825_A-0004 55
sw3505-A-0002 108
sw3505-A-0005 113
```

```
sw3689-B-0001 165
sw3689-B-0002 20
sw3689-B-0006 47
sw3822-B-0008 125
sw3824-B-0001 140
sw4093-A-0006 174
sw4141-B-0002 222
sw4322-A-0008 126
sw4373-A-0001 242
sw4373-A-0003 56
```

# Appendix B

# Sample Runs Through the System

## B.1 A Sample Run Through the Preprocessing Pipe

Explanation: Using a short example utterance the sample run shown in Figure prep-walk-thru demonstrates how the disfluent input is transformed into a "cleaned-up" version for the POS based chunk parser.

## B.2 A Sample Run Through the Whole System

To give a better idea what happens at the various stages of our system, I give commented excerpts from a log file from a system run that uses eight hypotheses from an interesting utterance, taken from the train set (en_6179_A-0011).[1]

```
Set of hypotheses (before the tagger):
********

you weren't born justice so cups on
you weren't born just to sew cups on
you weren't born justice vocal song
you weren't born just to soak up sun
you weren't foreign just to sew cups on
you weren't born justice so courts on
you weren't born just to sew carp song
you weren't boring just to soak up son
```

[1]Their hypothesis-numbers in the original Nbest-list are: 1, 3, 189, 190, 214, 269, 273, and 296.

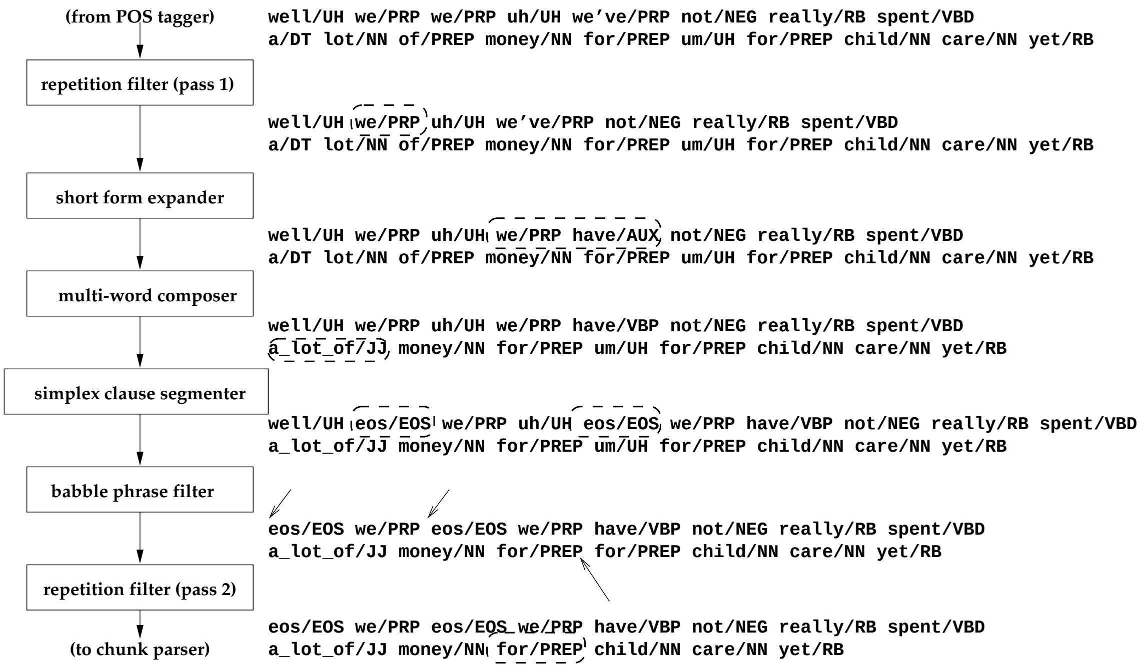


Figure B.1: An example walk through the preprocessing pipe.

```
Tagged hypotheses:
********

you/PRP weren't/AUX-N born/VBN justice/NN so/CC cups/NNS on/PREP
you/PRP weren't/AUX-N born/VBN just/RB to/TO sew/VB cups/NNS on/PREP
you/PRP weren't/AUX-N born/VBN justice/NN vocal/JJ song/NN
you/PRP weren't/AUX-N born/VBN just/RB to/TO soak/VB up/RP sun/NN
you/PRP weren't/AUX-N foreign/JJ just/RB to/TO sew/VB cups/NNS on/PREP
you/PRP weren't/AUX-N born/VBN justice/NN so/CC courts/NNS on/PREP
you/PRP weren't/AUX-N born/VBN just/RB to/TO sew/VB carp/NN song/NN
you/PRP weren't/AUX-N boring/JJ just/RB to/TO soak/VB up/RP son/NN

(... some preprocessing steps, in this case, not much happens ...)

Hypotheses after the simplex clause segmenter:
********

you/PRP weren't/AUX-N born/VBN justice/NN eos/EOS
so/CC cups/NNS on/PREP
you/PRP weren't/AUX-N born/VBN just/RB to/TO sew/VB cups/NNS on/PREP
you/PRP weren't/AUX-N born/VBN justice/NN vocal/JJ song/NN
you/PRP weren't/AUX-N born/VBN just/RB to/TO soak/VB up/RP sun/NN
you/PRP weren't/AUX-N foreign/JJ just/RB to/TO sew/VB cups/NNS on/PREP
you/PRP weren't/AUX-N born/VBN justice/NN eos/EOS
so/CC courts/NNS on/PREP
you/PRP weren't/AUX-N born/VBN just/RB to/TO sew/VB carp/NN song/NN
you/PRP weren't/AUX-N boring/JJ just/RB to/TO soak/VB up/RP son/NN

(... some other preprocessing steps ...)

Cleaned and combined output of the chunk parser:
********



#0
{you/PRP weren't/AUX-N born/VBN justice/NN eos/EOS }

[ _np ] ( you/PRP )
[ _vbneg ] ( weren't/AUX-N [ _vb_head ] ( born/VBN ) )
[ _np ] ( [ _n_head ] ( justice/NN ) )
[ _eos ] ( eos/EOS )

#1
{so/CC cups/NNS on/PREP }
```

```
[ _conj ] ( so/CC )
[ _np ] ( [ _n_head ] ( cups/NNS ) )
on/%PREP

#2
{you/PRP weren't/AUX-N born/VBN just/RB to/TO sew/VB cups/NNS on/PREP }


[ _np ] ( you/PRP )
[ _vbneg ] ( weren't/AUX-N [ _vb_head ] ( born/VBN ) )
[ _adv ] ( just/RB )
[ _toinf ] ( to/TO [ _vb_head ] ( sew/VB ) )
[ _np ] ( [ _n_head ] ( cups/NNS ) )
on/%PREP

#3
{you/PRP weren't/AUX-N born/VBN justice/NN vocal/JJ song/NN }


[ _np ] ( you/PRP )
[ _vbneg ] ( weren't/AUX-N [ _vb_head ] ( born/VBN ) )
[ _np ] ( [ _n_head ] ( justice/NN ) [ _adjp ] ( vocal/JJ ) [ _n_head ] ( song/NN ) )

(... score calculation and rescoring ...)

New ranks from NN-rescoring:
----------------------------
8/7/4/3/6/5/1/2
WER_first_old - WER_first_new = 0.625-0.25 = 0.375 (=WER gain)

Score-Table:
------------
Hypo-Rank  True WER  Chunk-Cov. Non-parsed  Skipped  Chunk-LM  Norm.SR
New/Old              Score      Words       Sect.     Score    Score
-------------------------------------------------------------------
1/8     ** 0.25        0.875      0          0          0.984     0.93
2/7        0.375       0.625      0          0          0.865     0.94
3/4        0.0         0.75       0          0          0.954     0.97
4/3        0.625       0.5        0          0          0.618     0.98
5/6        0.625       0.625      0.125      0.125      0.715     0.95
6/5        0.5         0.75       0.125      0.125      1.056     0.96
7/1     ** 0.625       0.625      0.125      0.125      0.715     1.0
8/2        0.375       0.625      0.125      0.125      1.032     0.99
```