Dependency Parsing with an Extended Finite State Approach

Kemal Oflazer
Computing Research Laboratory
New Mexico State University,
Las Cruces, NM, 88001
and
Department of Computer Engineering
Bilkent University
Bilkent, Ankara, 06533,Turkey
ko@crl.nmsu.edu

January 22, 1999

Abstract

This paper presents a dependency parsing scheme using an extended finite state approach. The parser augments input representation with "channels" so that links representing syntactic dependency relations among words can be accommodated, and iterates on the input a number of times to arrive at a fixed point. Intermediate configurations violating various constraints of projective dependency representations such as no crossing links, no independent items except sentential head, etc, are filtered via finite state filters. We have applied the parser to dependency parsing of Turkish.

1 Introduction

Recent advances in the development of sophisticated tools for building finite state systems (e.g., XRCE Finite State Tools (Karttunen et al., 1996), AT&T Tools (Mohri et al., 1998)) have fostered the development of quite complex finite state systems for natural language processing. In the last few years, there have been many studies on developing finite state parsing systems. Koskenniemi and his colleagues (1990; 1992) have used finite state techniques for parsing by reductionistic constraints. Recently Ait-Mokthar and Chanod (1997) has presented a finite state parser which incrementally revises the syntactic representation it generates. Grefetstette (Grefenstette, 1996) has used finite state techniques for constituent bracketing and extracting grammatical relations. There have also been a number of approaches to natural language parsing using extended finite state approaches in which a finite state engine is applied multiple times to the input, or various derivatives thereof, until some stopping condition is reached. Roche (1997) presents an approach for parsing in which the input is iteratively bracketed using a finite state transducer. Abney(1996) presents a finite state parsing approach in which a tagged sentence is parsed by transducers which progressively transform the input to sequences of symbols representing phrasal constituents. This paper presents an approach to dependency parsing using an extended finite state model resembling the approaches of Roche and Abney. The parser produces outputs that encode a labeled dependency tree representation of the syntactic relations between the words in the sentence.

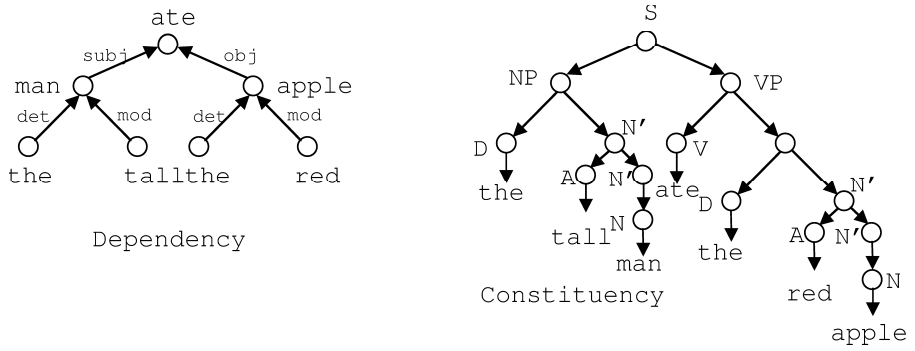


Figure 1: Dependency vs. Constituency Representations for *The tall man ate the red apple.*

The outline of the paper is as follows. In Section 2 reviews the salient points of dependency syntax. Section 3 briefly discusses aspects of Turkish relevant for this work. Section 4 very briefly summarizes basic relevant concepts on finite state transducers. Section 5 presents our approach to dependency parsing with an extended finite state approach. Section 6 presents some very preliminary results from our experiments with parsing Turkish sentences. Finally, Section 7 discusses some extensions and future work.

2 Dependency Syntax

Dependency approaches to syntactic representation use the notion of syntactic relation to associate surface lexical items. Figure 1 shows dependency and constituency representations for a simple English sentence.[1] The book by Melčuk (1988) presents a comprehensive exposition of dependency syntax. Computational approaches to dependency syntax have recently become quite popular (e.g., a workshop dedicated to computational approaches to dependency grammars has been held at COLING/ACL'98 Conference). Järvinen and Tapananinen have demonstrated an efficient wide-coverage dependency parser for English (Tapanainen and Järvinen, 1997; Järvinen and Tapanainen, 1998). The work of Sleator and Temperley(1991) on *link grammar*, an essentially lexicalized variant of dependency grammar, has also proved to be interesting in a number of aspects. Dependency-based statistical language modeling and analysis have also become quite popular in statistical natural language processing (Lafferty et al., 1992; Eisner, 1996; Chelba and et al., 1997).

Robinson(1970) gives four axioms for well-formed dependency structures, which have been assumed in almost all computational approaches. In a dependency structure of a sentence (i) one and only one word is independent, i.e., not linked to some other word, (ii) all others depend directly on some word, (iii) no word depends on more than one other, and, (iv) if a word A depends directly on B, and some word C intervenes between them (in linear order), then C depends directly on A or on B, or on some other intervening word. This last condition of

[1] We draw dependency arcs from the dependent to the head.

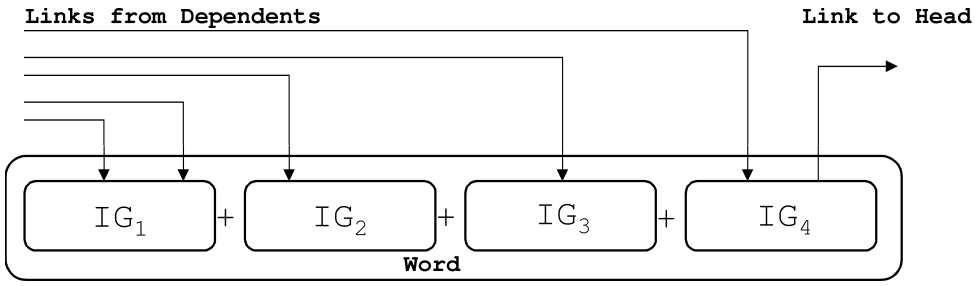


Figure 2: Links and Inflectional Groups

projectivity (or various extensions of it; see e.g., Lau and Huang (1994)) is usually assumed by most computational approaches to dependency grammars as a constraint for filtering configurations. It has also been used as a simplifying condition in statistical approaches for inducing dependencies from corpora (e.g., Yüret(1998).)

3 Turkish

Turkish is an agglutinative language where a sequence of inflectional and derivational morphemes get affixed to a root (Oflazer, 1993). Derivations are very productive, and the syntactic relations that a word is involved in as a dependent or head element, are determined by the inflectional properties of the one or more (intermediate) derived forms. In this work, we assume that a Turkish word is represented as a sequence of *inflectional groups* (IGs hereafter), separated by `^DB`s denoting derivation boundaries, in the following general form:

$\texttt{root+Infl}_1\texttt{\^{}DB+Infl}_2\texttt{\^{}DB+}\cdots\texttt{\^{}DB+Infl}_n$

where $\texttt{Infl}_i$ denote relevant inflectional features including the part-of-speech for the root or any of the derived forms. For instance, the derived determiner `sağlamlaştırdığımızdaki` (literally, the (one) on the one we caused to become strong) would be represented as:[2]

```
sağlam+Adj^DB+Verb+Become^DB+Verb+Caus+Pos^DB+Adj+PastPart+P1sg^DB
    +Noun+Zero+A3sg+Pnon+Loc^DB+Det
```

This word has 6 IGs:

1. `sağlam+Adj`
2. `+Verb+Become`
3. `+Verb+Caus+Pos`
4. `+Adj+PastPart+P1sg`
5. `+Noun+Zero+A3sg+Pnon+Loc`
6. `+Det`

An interesting observation that we can make about Turkish is that, when a word is considered as a sequence of IGs, syntactic relation links only emanate from the last IG of a (dependent) word, and land on one of the IG's of the (head) word on the right (with minor exceptions), as exemplified in Figure 2. A second observation is that, with minor exceptions, the dependency links between the IGs, when drawn above the IG sequence, do not cross. Figure 3 shows a dependency tree for the following sentence laid on top of the words segmented along IG

[2]The morphological features other than the obvious POSs are:`+Become`: become verb,`+Caus`: causative verb, `PastPart`:Derived past participle, `P1sg`: 1sg possessive agreement, `A3sg`: 3sg number-person agreement,`+Zero`:Zero derivation with no overt morpheme, `+Pnon`: No possessive agreement, `+Loc`:Locative case, `+Pos`: Positive Polarity.

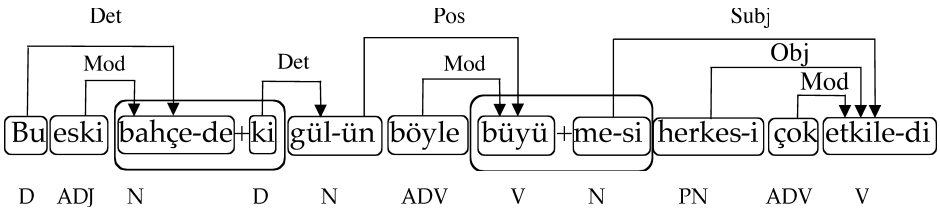


Figure 3: Dependency Links in an example Turkish Sentence

boundaries.

(1) Bu eski bahçe-de+ki gül-ün

`bu(this)+Det old+Adj bahçe(garden)+A3sg+Pnon+Locˆ+Det gül(rose)+Noun+A3sg+Pnon+Gen`

The growth of the rose

böyle büyü+me-si

`böyle(like-this)+Adv büyü(grow)+Verb+Posˆ DB+Noun+Inf+A3sg+P3sg+Nom`

like this in this old garden impressed everybody.

herkes-i çok etkile-di.

`herkes(everybody)+Pron+A3sg+Pnon+Acc çok(very)+Adv etkile(impress)+Verb+Pos+Past+A3sg`

4 Finite State Transducers

Finite state transducers (FST hereafter) are finite state devices that map between two regular languages U and L (Kaplan and Kay, 1994; Mohri, 1996; Mohri, 1997). Figure 4 summarizes the basic ideas of a FST. The transitions of a FST are labeled with symbol pairs $u : l$, either, but not both of which, can be the ϵ symbol. The symbol u is the "upper" symbol belonging to the alphabet of the "upper"language, and l is the "lower" symbol belonging to the alphabet of the "lower" regular language. FSTs are defined by regular expressions over such pairs of symbols.[3] An example of such regular expression is given on the upper right side of Figure 4. Clearly, such description machinery for finite state transductions are too low level, and one needs higher level notations to describe complex operations. The expression on the lower right side of the same figure use the language primitives of XRCE Finite State Tools (Karttunen et al., 1996), and describes a transducer which inserts the symbols `"[NP"` and `"NP]"`, around patterns in the upper language which match the regular expression `D A* N`.

A important operation on FSTs that we will be referring to in the following sections is the *composition* operation. Let T_1 and T_2 be two transducers mapping between upper and lower

[3] When both the upper and lower symbols are the same, one of them suffices notationally.

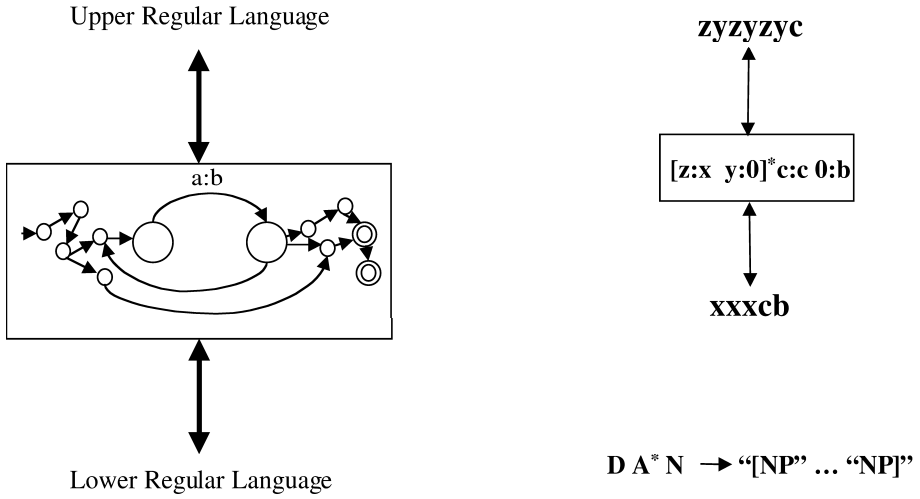


Figure 4: Finite State Transducers

languages U_1 and L_1, and U_2 and L_2 respectively. Then, the composed transducer $T = T_1 \circ T_2$ maps between $U = T_1^{-1}(L_1 \cap U_2)$ and $L = T_2(L_1 \cap U_2)$.[4]

5 Finite State Dependency Parsing

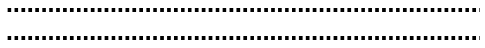
Our approach relies on augmenting the input with "channels" that (logically) reside above the IG sequence and "laying" links representing dependency relations in these channels, as depicted Figure 5 a). The parser operates in a number of iterations: At each iteration of the parser, an empty channel is added to the input and any possible links are established using these channels, until no new links can be added. An abstract view of this is presented in parts b) through e) of Figure 5.

5.1 Representing Channels and Syntactic Relations

The sequence (or the chart) of IGs is produced by a FST incorporating a morphological analyzer, with each IG being augmented by two pairs of delimiter symbols, as `<(IG)>`. Word final IGs (WFIG), IGs that links will emanate from, are further augmented with a special marker `@`.

[4] T^{-1} stands for the reverse transduction from L to U.

a) Input sequence of IGs are augmented with symbols to represent Channels.

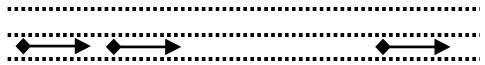


$(IG_1)\ (IG_2)\ (IG_3)...\ (IG_i)...\ (IG_{n-1})\ (IG_n)$

b) Links are embedded in channels.

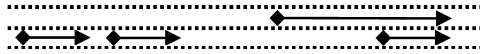
$(IG_1)\ (IG_2)\ (IG_3)...\ (IG_i)...\ (IG_{n-1})\ (IG_n)$

c) New channels are "stacked on top of each other".

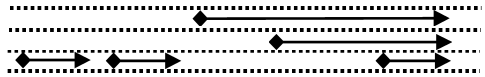


$(IG_1)\ (IG_2)\ (IG_3)...\ (IG_i)...\ (IG_{n-1})\ (IG_n)$

d) So that links that can not be accommodated in lower channels can be established.



$(IG_1)\ (IG_2)\ (IG_3)...\ (IG_i)...\ (IG_{n-1})\ (IG_n)$



$(IG_1)\ (IG_2)\ (IG_3)...\ (IG_i)...\ (IG_{n-1})\ (IG_n)$

Figure 5: Channels and Links

Channels are represented by pairs of matching symbols that surround the `<...(` and the `)...>` pairs. Channels are inserted by a FST as shown in Figure 6, which depicts the initial insertion. Symbols for new channels (upper channels in Figure 5) are stacked so that the symbols for the most recent channels are those closest to the `(...)`.[5] The symbol `0` indicates that the channel segment is not used while `1` indicates that the channel is used by a link that starts at some IG on the left and ends at some IG on the right. If a link starts from an IG (ends on an IG), then a start (stop) symbol denoting the syntactic relation is used on the right (left) side of the IG. The syntactic relations (along with symbols used) that we currently encode in our parser are the following:[6]

1. `S`: Subject
2. `O`: Object,
3. `M`: Modifier (adverbs/adjectives)
4. `P`: Possessor,
5. `C`: Classifier
6. `D`: Determiner
7. `T`: Dative Adjunct
8. `L`: Locative Adjunct
9. `A`: Ablative Adjunct
10. `I`: Instrumental Adjunct

For instance, with three channels, two IGs of the *bahçedeki* in Figure 3, would be represented

[5] At any time, the number of channel symbols on both sides of an IG are the same.

[6] We use the lower case symbol to mark the start of the link and the upper case symbol to encode the end of the link.

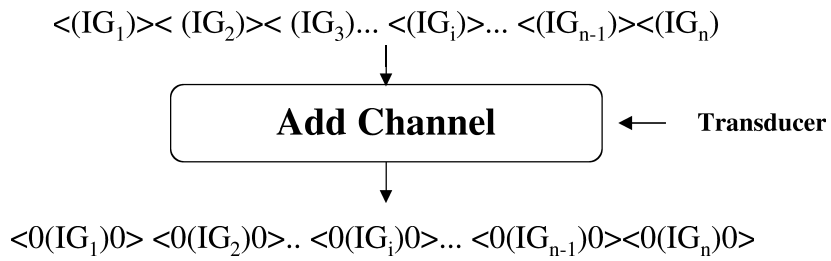


Figure 6: Inserting Empty Channels

as `<MD0(bahçe+Noun+A3sg+Pnon+Loc)000> <000(+Det@)00d>`

5.2 Components of a Parser Stage

The basic strategy of a parser stage is to recognize by a rule (encoded as a regular expression) a dependent IG and a head IG and link them by modifying the "topmost" channel between those two. To achieve this:

1. We put temporary brackets to the left of the dependent IG and to the right of the head IG, making sure that (i) the last channel in that segment is free, and (ii) the dependent is not already linked (at one of the lower channels.)
2. We mark the channels of the start, intermediate and ending IGs with the appropriate symbols encoding the relation thus established by the brackets, and
3. We remove the temporary brackets.

A typical linking rule looks like the following:[7]

```
[LL IG1 LR] [ML IG2 MR]* [RL IG3 RR] (->) "{S"  "S}"
```

This rule says: *(optionally) bracket (with* `{S` *and* `S}`*), any occurrence of morphological pattern IG1 (dependent), followed by any number of occurrences of pattern IG2, finally ending with a pattern IG3 (governor).* The symbols `L(eft)L(eft)`, `LR`, `ML`, `MR`, `RL` and `RR` are regular expressions that encode constraints on the bounding channel symbols. For instance, `LR` is the pattern `"@" ")" "0" ["0" | 1]* ">"` which checks that (i) this is a WFIG is word-final(has a `"@"`), (ii) the right side "topmost" channel is empty (channel symbol nearest to `")"`is `"0"`), and (iii) the IG is not linked to any other in the lower channels (the only symbols on the right side are `0`s and `1`s.)

An example of a simple rule which links a dative case marked nominal to the immediately following postposition that subcategorizes for a dative object is the following:

[7] We use the XRCE Regular Expression Language Syntax; see `http://www.xrce.xerox.com/research/-mltt/fst/fssyntax.html` for details.

```
[LL  DativeNominal LR] [RL DativePostPos RR] (->) "{O"  "O}" }
```

where `DativeNominal` and `DativePostPos` are regular expressions for morphological patterns matching dative nominal morphological parses (nouns, pronouns, infinitives, nominal participles, etc.) and morphological patterns for dative requiring postpositions, respectively.

```
    [LL  NominativeNominalA3pl LR] [ML AnyIG MR]* \\\hspace*{1cm} \hspace*{1cm} [RL
[FiniteVerbA3sg  | FiniteVerbA3pl] RR ](->) "\{S"  "S\}"
```

is used to bracket a segment starting with a plural nominative nominal, as subject of a finite verb on the right with either `+A3sg` or `+A3pl` number person agreement (allowed in Turkish.) The regular expression `NominativeNominalA3pl` matches any nominal IG with nominative case and `A3pl` plural agreement, while the regular expression `[FiniteVerbA3sg | FiniteVerbA3pl]` matches any finite verb IG with either `A3sg` or `A3pl` agreement. The regular expression `AnyIG` matches any IG.

All the rules are grouped together into a parallel bracket rule defined as follows:

```
define Bracket [
    Pattern1 (->) "{1" . . ."1}",
    Pattern2 (->) "{2" . . ."2}",
    Pattern3 (->) "{3" . . ."3}",
    . . .
    PatternN (->) "{n" . . ."n}"
];
```

which will produce all possible bracketing of the input IG sequence.[8]

5.3 Filtering Crossing Link Configurations

The bracketings produced by `Bracket` contain configurations that may cause crossing links. This happens when the left side channel symbols of the IG immediately right of a open bracket contains the symbol `1` for one of the lower channels, indicating a link entering the region, or when the right side channel symbols of the IG immediately to the left of a close bracket contains the symbol `1` for one of the lower channels, indicating a link exiting the segment, i.e., either or both of the following patterns appear in the bracketed segment:

`{S < ... 1 ... 0 ( ... )  ...` or `) 0 ... 1 ... > S}`

Configurations generated by bracketing are filtered by FSTs implementing suitable regular expressions that reject inputs having crossing links.

A second configuration that may appear is the following: A rule may attempt to put a link in the topmost channel even though the corresponding segment is not utilized in a previous

[8] $\{_i$ and $_i\}$ are pairs of brackets; there is a distinct pair for each syntactic relation to be identified.

channel, e.g., the one previous channel may be all 0s. This constraint filters such cases to prevent redundant configurations from proliferating for later iterations of the parser.[9]

For these two configuration constraints we define `FilterConfigurations`

```
define FilterConfigurations [ FilterCrossingLinks .o.  FilterEmptySegments];}
```

We can now define one phase (of one iteration) of the parser as:

```
define Phase          Bracket  .o.
              FilterConfigurations .o.
                  MarkChannels .o.
              RemoveTemporaryBrackets;
```

The transducer `MarkChannels` modifies the channel symbols in the bracketed segments to either the syntactic relation start and end symbols or a `1`. Finally, the transducer `RemoveTemporaryBrackets`, removes the brackets.

The formulation up to does not allow us to bracket an IG on two consecutive non-overlapping links in the same channel. We would need a bracketing configuration like `...{S <...> ...{M <...> S}...<...>M} ...` but this would not be possible within `Bracket`, as patterns check that no other brackets are within their segment of interest. Simply composing the `Phase` transducer with itself without introducing a new channel solves this problem, giving us a one-stage parser, i.e., `define Parse Phase .o.  Phase;`

5.4 Enforcing Syntactic Constraints

The rules linking the IGs are overgenerating in that they may generate configurations that may violate some general or language specific constraints. For instance, more than one subject or one object may attach to a verb, or more that one determiner or possessor may attach to a nominal, an object may attach to a passive verb (conjunctions are handled in the manner described in Järvinen and Tapanainen(1998)), or a nominative pronoun may be linked as a direct object (which is not possible in Turkish), etc. Constraints preventing these may can be encoded in the bracketing patterns, but doing so results in complex and unreadable rules. Instead, each can be implemented as finite state filters which operate on the outputs of `Parse` by checking the symbols denoting the relations. For instance we can define the following regular expression for filtering out configurations where two determiners are attached to the same IG.

```
define AtMostOneDet   [ "<" [ ~[[$"D"]^1] & LeftChannelSymbols* ]
                                  "(" AnyIG ("@") ")"
                                            RightChannelSymbols* ">" ]*;
```

[9]This constraint is a bit trickier since one has to check that the same number of channels on both sides are empty; we limit ourselves to the last 3 channels in the implementation.

The FST for this regular expression makes sure that all configurations that are produced have at most one D symbol among the left channel symbols.[10] Other constraints (for instance, no objects for passive marked verbs) can check any subsequent IG patterns, and kill a configuration if a passive IG is found.

Once all such constraints (that we will assume will be labeled as $\texttt{Cons}_1$, $\texttt{Cons}_2$ ...$\texttt{Cons}_n$), they can be composed to give one FST that enforces all of these:

```
define SyntacticFilter [Cons1 .o.  Cons2 .o.  ... .o.  Consn];
```

5.5 Iterative application of the parser

Full parsing consists of iterative applications of the Parser and SyntacticFilter FSTs. Let Input be a transducer that represents the word sequence. Let LastChannelNotEmpty be a transducer which detects if any configuration has at least one link established in the last channel added, defined as follows:

```
define LastChannelNotEmpty ["<" LeftChannelSymbols+ "(" AnyIG ("@") ")"
                                        RightChannelSymbols+ ">"]* -
                           ["<" LeftChannelSymbols*  0 "(" AnyIG ("@") ")"
                                   0 RightChannelSymbols* ">"]*;}
```

MorphologicalDisambiguator is a reductionistic finite state disambiguator which performs accurate but very conservative local disambiguation and multi-word construct coalescing, to reduce morphological ambiguity without making any errors.

The iterative applications of the parser can now be given (in pseudo-code) as:

```
/*     Map words to a transducer representing a chart of IGs */
       M = [Input .o. MorphologicalAnalyzer] .o.  MorphologicalDisambiguator;
       repeat {
            M = M .o. AddChannel .o. Parse .o. SyntacticFilter;
       }
       until  ( [M .o. LastChannelNotEmpty].l == { })
       M = M .o. OnlyOneUnlinked ;
       Parses = M.l;
```

This procedure iterates until the most recently added channel of every configuration generated is unused (i.e., the (lower regular) language recognized by M .o. LastChannelNotEmpty is

[10] The crucial portion at the beginning says "it is not the case that there is more than one substring containing D."

empty.)

The step after the loop, `M = M .o. OnlyOneUnlinked`, enforces the constraint that in a correct dependency parse all except one of the word final IGs have to link as a dependent to some head. This transduction filters all those configurations (and usually there are many of them due to the optionality in the bracketing step.) Then, `Parses` defined as the (lower) language of the resulting FST has all the strings that encode the IGs and the links.

5.6 Robust Parsing

It is possible that either because of grammar coverage, or ungrammatical input, a parse with only one unlinked WFIG may not be found. In such cases `Parses` above would be empty. One may however opt to accept parses with $k > 1$ unlinked WFIGs when there are no parses with $\leq k$ unlinked WFIGs (for some small k.) This can be achieved by using the *lenient composition operator* (Karttunen, 1998). Lenient composition, notated as `.O.`, is used with a *generator–filter* combination. When a generator transducer `G` is leniently composed with a filter transducer, `F`, the resulting transducer, `G .O. F`, has the following behavior when an input is applied: If any of the outputs of `G` in response to the input string satisfies the filter `F`, then `G .O. F` produces just these as output. Otherwise, `G .O. F` outputs what `G` outputs. Karttunen originally used lenient composition in an elegant formulation of constraint ranking in optimality theory (Karttunen, 1998) which also involved selecting parses will smaller violations of the constraints. It is this latter application of lenient composition that we can import into our formulation.

Let `Unlinked_`i denote a regular expression which accepts parse configurations with less than or equal i unlinked word-final IGs. For instance for $i = 2$, this would be defined as follows:

```
~[[$[ "<" LeftChannelSymbols* "(" AnyIG  "@"  ")" ["0" |   1]* ">"]]^ >  2  ]};
```

This regular expression will accept only those outputs where the number of unlinked WFIGs not greater than 2.

Replacing line `M = M .o. OnlyOneUnlinked`, with, for instance, `M = M .O. Unlinked_1 .O. Unlinked_2 .O. Unlinked_3;` will have the parser produce outputs with up to 3 unlinked WFIGs, when there are no outputs with a smaller number of unlinked WFIGs. Thus it is possible to recover some of the partial dependency structures when a full dependency structure is not available for some reason. The caveat would be however that since `Unlinked_1` is a very strong constraint, any relaxation would increase the number of outputs substantially.

6 Preliminary experiments with dependency parsing of Turkish

Our work to date has mainly consisted of developing and implementing the representation and finite state techniques involved here, along with a non-trivial grammar component. At this

Avg. Words/Sentence:	11.6 (Min=4 – Max=23)
Avg. IGs/Sentence:	16.6 (5 – 36)
Avg. Parser Iterations:	5.3 (3 – 8)
Avg. Parses/Sentence:	28.35 (2 – 132)

Table 1: Preliminary Statistics from Parsing

point, we have not done any large scale experimentation on coverage, but rather restricted our attention to a small corpus of 20 sentences (some of which are quite complex) from Turkish news text.

The grammar has two major components. The morphological analyzer is a full coverage analyzer built using XRCE tools, slightly modified to generate outputs as a sequence of IGs for a sequence of words. When an input sentence is (again represented as a transducer denoting a sequence of words) is composed with the morphological analyzer (see pseudo-code above), a transducer for the chart representing all IGs for all morphological ambiguities (remaining after disambiguation) is created. The dependency relations are described by a set of about 30 patterns much like the ones exemplified above. The rules are almost all non-lexical establishing links of the types listed earlier. Conjunctions are handled by linking the left conjunct to the conjunction, and linking the conjunction to the right conjunct (possibly at a different channel). There are an additional set of about 25 finite state constraints that impose various syntactic and configurational constraints. The resulting `Parser` transducer had 2707 states 27,713 transitions while the `SyntacticConstraints` transducer had 28,894 states and 302,354 transitions.

Table 1 presents our preliminary results for parsing our corpus. Although these results are very preliminary, we are encouraged by the approach and the results. The finite state transducers compile in about 2 minutes on Apple Macintosh 250 Mhz Powerbook. Parsing is about a second per iteration including lookup in the morphological analyzer. With completely morphologically disambiguated input, parsing is instantaneous. The number of iterations also count the last iteration where no new links are added.

Figure 7 presents the input and the output of the parser for a sample Turkish sentence. The output of the parser is processed with a Perl script to provide a more human-consumable presentation:

7 Discussion and Conclusions

We have presented the architecture and implementation of novel extended finite state dependency parser, with preliminary results from Turkish. We have formulated, but not yet implemented at this stage, two extensions. Crossing dependency links are very rare in Turkish and almost always occur in Turkish when an adjunct of a verb cuts in a certain position of a (dis-

continuous) noun phrase. We can solve this by allowing such adjuncts to use a special channel "below" the IG sequence so that limited crossing link configurations can be allowed. Links where the dependent is to the right of its head, which can happen with some of the word order variations allowed in Turkish, can similarly be handled with a right-to-left version of `Parser` which is applied during each iteration.

In addition to the reductionistic disambiguator that we have used just prior to parsing, we have implemented a number of heuristics to limit the number of potentially spurious configurations that result because of optionality in bracketing, mainly by enforcing obligatory bracketing from sequential dependency configurations (e.g., the complement of a postposition is immediately before it.) Such heuristics force such dependencies to appear in the first channel at the first possible chance and hence prune many potentially useless configurations popping up in later stages. The robust parsing technique has been very instrumental during the process mainly in the debugging of the grammar, but we have not made any substantial experiments with it yet.

Our ongoing work is on extending both the formulation of the approach to cover limited forms of crossing links, in increasing the coverage of the grammar. We expect to present a more comprehensive evaluation in the final version.

8 Acknowledgments

This work was partially supported by a NATO Science for Stability Program Project Grant, TU-LANGUAGE made to Bilkent University. A portion of this work was done while the author was visiting Computing Research Laboratory at New Mexico State University. The author thanks Lauri Karttunen of Xerox Research Centre Europe, Grenoble for making available XRCE Finite State Tools.

References

Steven Abney. 1996. Partial parsing via finite state cascades. In *Proceedings of the ESSLLI'96 Robust Parsing Workshop.*

Salah Ait-Mokhtar and Jean-Pierre Chanod. 1997. Incremental finite-state parsing. In *Proceedings of ANLP'97*, pages 72 – 79, April.

Ciprian Chelba and et al. 1997. Structure and estimation of a dependency language model. In *Processings of Eurospeech'97.*

Jason Eisner. 1996. Three new probabilistic models for dependency parsing: An exploration. In *Proceedings of the 16th International Conference on Computational Linguistics (COLING-96)*, pages 340–345, August.

Gregory Grefenstette. 1996. Light parsing as finite-state filtering. In *ECAI '96 Workshop on Extended finite state models of language.* August.

Timo Järvinen and Pasi Tapanainen. 1998. Towards an implementable dependency grammar. In *Proceedings of COLING/ACL'98 Workshop on Processing Dependency-based Grammars*, pages 1–10.

Ronald M. Kaplan and Martin Kay. 1994. Regular models of phonological rule systems. *Computational Linguistics*, 20(3):331–378, September.

Lauri Karttunen, Jean-Pierre Chanod, Gregory Grefenstette, and Anne Schiller. 1996. Regular expressions for language engineering. *Natural Language Engineering*, 2(4):305–328.

Lauri Karttunen. 1998. The proper treatment of optimality theory in computational linguistics. In Lauri Karttunen and Kemal Oflazer, editors, *Proceedings of the International Workshop on Finite State Methods in Natural Language Processing–FSMNLP*, June.

Kimmo Koskenniemi, Pasi Tapanainen, and Atro Voutilainen. 1992. Compiling and using finite-state syntactic rules. In *Proceedings of the 14th International Conference on Computational Linguistics, COLING-92*, pages 156–162.

Kimmo Koskenniemi. 1990. Finite-state parsing and disambiguation. In *Proceedings of the 13th International Conference on Computational Linguistics, COLING'90*, pages 229 – 233.

John Lafferty, Daniel Sleator, and Davy Temperley. 1992. Grammatical trigrams: A probabilistic model of link grammars. In *Proceedings of the 1992 AAAI Fall Symposium on Probablistic Approaches to Natural Language.*

Bong Yeung Tom Lai and Changning Huang. 1994. Dependency grammar and the parsing of Chinese sentences. In *Proceedings of the 1994 Joint Conference of 8th ACLIC and 2nd PaFoCol.*

Igor A. Melčuk. 1988. *Dependency Syntax: Theory and Practice.* State University of New York Press.

Mehryar Mohri, Fernando Pereira, and Michael Riley. 1998. A rational design for a weighted finite–state transducer library. In *Lecture Notes in Computer Science, 1436.* Springer Verlag.

Mehryar Mohri. 1996. On some applications of finite-state automata theory to natural language processing. *Natural Language Engineering*, 2:1–20.

Mehryar Mohri. 1997. Finite-state transducers in language and speech processing. *Computational Linguistics*, 23(2):269–311, June.

Kemal Oflazer. 1993. Two-level description of Turkish morphology. In *Proceedings of the Sixth Conference of the European Chapter of the Association for Computational Linguistics*, April. A full version appears in *Literary and Linguistic Computing*, Vol.9 No.2, 1994.

Jane J. Robinson. 1970. Dependency structures and transformational rules. *Language*, 46(2):259–284.

Emmanuel Roche. 1997. Parsing with finite state transducers. In Emmanuel Roche and Yves Schabes, editors, *Finite–State Language Processing*, chapter 8. The MIT Press.

Daniel Sleator and Davy Temperley. 1991. Parsing English with a link grammar. Technical Report CMU-CS-91-196, Computer Science Department, Carnegie Mellon University.

Pasi Tapanainen and Timo Järvinen. 1997. A non-projective dependency parser. In *Proceedings of ANLP'97*, pages 64 – 71, April.

Deniz Yüret. 1998. *Discovery of Linguistic Relations Using Lexical Attraction.* Ph.D. thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology.

Input Sentence:

Dünya BankasıTürkiye Direktörü hükümetin izlediği ekonomik programın sonucunda önemli adımların atıldığını söyledi.

(Word Bank Turkey Director said that as a result of the economic program followed by the government, important steps were taken.)

Parser Output after 3 iterations:

```
Parse1:

<000(dUnya+Noun+A3sg+Pnon+Nom@)00c><C00(banka+Noun+A3sg+P3sg+Nom@)0c0>
<010(tUrkiye+Noun+Prop+A3sg+Pnon+Nom@)01c><CC0(direktOr+Noun+A3sg+P3sg+Nom@)s00>
<001(hUkUmet+Noun+A3sg+Pnon+Gen@)10s><S01(izle+Verb+Pos)100><001(+Adj+PastPart+P3sg@)1m0>
<011(ekonomik+Adj@)11m><MM1(program+Noun+A3sg+Pnon+Gen@)10p><P01(sonuC+Noun+A3sg+P3sg+Loc@)1l0>
<011(Onem+Noun)110><011(+Adj+With@)11m><M11(adIm+Noun+A3pl+Pnon+Gen@)11s>
<S11(at+Verb)110><011(+Verb+Pass+Pos)110><011(+Noun+PastPart+A3sg+P3sg+Acc@)11o>
 ***
<OLS(sOyle+Verb+Pos+Past+A3sg@)000>
 ***
Parse2:

<000(dUnya+Noun+A3sg+Pnon+Nom@)00c><C00(banka+Noun+A3sg+P3sg+Nom@)0c0>
<010(tUrkiye+Noun+Prop+A3sg+Pnon+Nom@)01c><CC0(direktOr+Noun+A3sg+P3sg+Nom@)s00>
<001(hUkUmet+Noun+A3sg+Pnon+Gen@)10s><S01(izle+Verb+Pos)100><001(+Adj+PastPart+P3sg@)1m0>
<011(ekonomik+Adj@)11m><MM1(program+Noun+A3sg+Pnon+Gen@)10p><P01(sonuC+Noun+A3sg+P3sg+Loc@)1l0>
<011(Onem+Noun)110><011(+Adj+With@)11m><M11(adIm+Noun+A3pl+Pnon+Gen@)11s>
<SL1(at+Verb)100><001(+Verb+Pass+Pos)100><001(+Noun+PastPart+A3sg+P3sg+Acc@)10o>
 ***
<OOS(sOyle+Verb+Pos+Past+A3sg@)000>
 ***
```

The only difference in the two are parses are in the locative adjunct attachment (to verbs *at-* and *söyle*, highlighted with ***).

Dependency tree for the second parse:

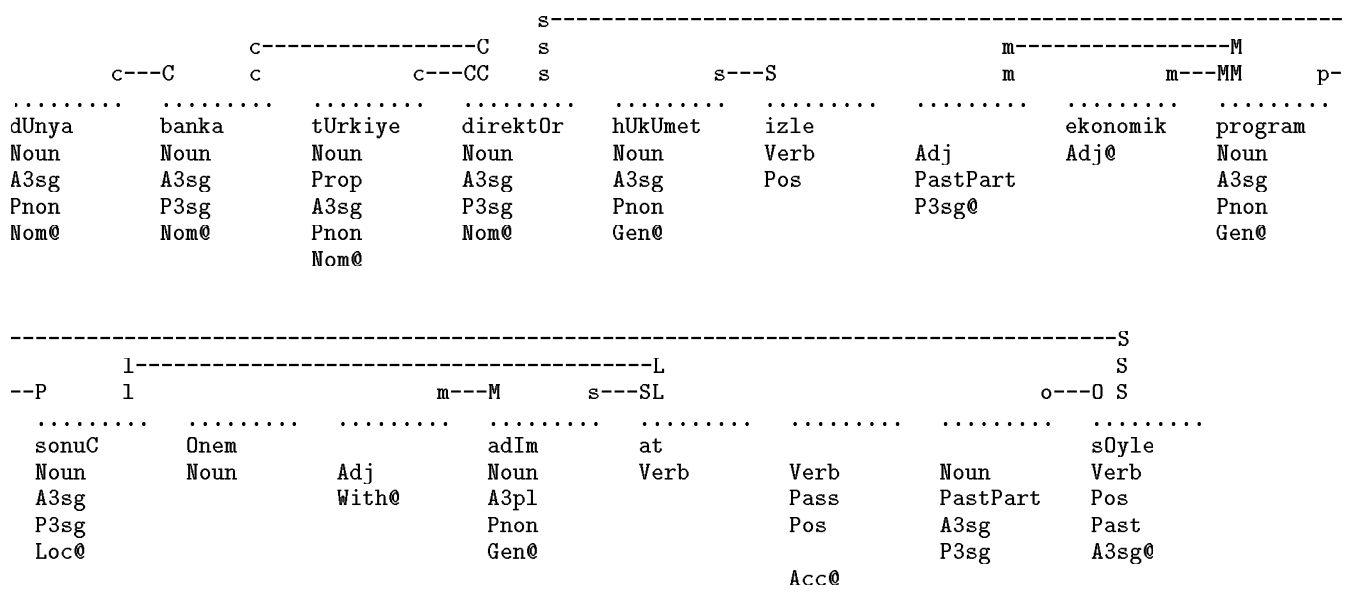


Figure 7: Sample Input and Output of the parser