Dependency Parsing with an Extended Finite State Approach

Kemal Oflazer

Department of Computer Engineering
Bilkent University
Ankara, 06533,Turkey
ko@cs.bilkent.edu.tr

Computing Research Laboratory
New Mexico State University
Las Cruces, NM, 88003 USA
ko@crl.nmsu.edu

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 several years, there have been a number of studies on developing finite state parsing systems, (Koskenniemi, 1990; Koskenniemi et al., 1992; Grefenstette, 1996; Ait-Mokhtar and Chanod, 1997). 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.

We assume that the reader is familiar with the basic concepts of finite state transducers (FST hereafter), finite state devices that map between two regular languages U and L (Kaplan and Kay, 1994).

2 Dependency Syntax

Dependency approaches to syntactic representation use the notion of syntactic relation to associate surface lexical items. 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 Tapanainen 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 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, and 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

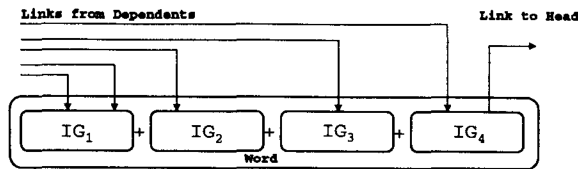


Figure 1: Links and Inflectional Groups

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 ^DBs denoting derivation boundaries, in the following general form:

root+Infl$_1$^DB+Infl$_2$^DB+···^DB+Infl$_n$

where 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**[1] 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

A sentence would then be represented as a sequence of the IGs making up the words.

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 1. A second observation is that, with minor exceptions, the dependency links between the IGs, when drawn above the IG sequence, do not cross. Figure 2 shows a dependency tree for a sentence laid on top of the words segmented along IG boundaries.

4 Finite State Dependency Parsing

The 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 3 a). The parser operates in a number of iterations: At each iteration of the parser, an new empty channel

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 3: Channels and Links

is "stacked" on top of 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 3.

4.1 Representing Channels and Syntactic Relations

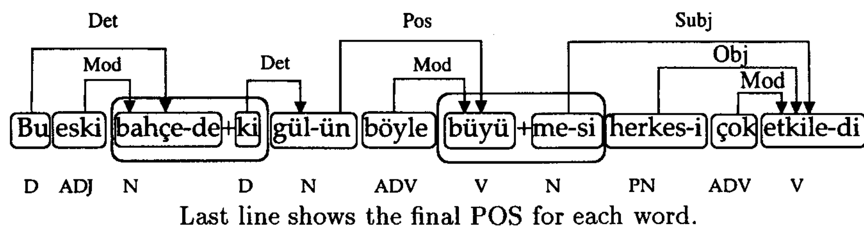
The sequence (or the chart) of IGs is produced by a a morphological analyzer FST, with each IG being augmented by two pairs of delimiter symbols, as <(IG)>. Word final IGs, IGs that links will emanate from, are further augmented with a special marker @. Channels are represented by pairs of matching symbols that surround the <...(and the)...> pairs. Symbols for new channels (upper channels in Figure 3) are stacked so that the symbols for the topmost channels are those closest to the (...).[3] The channel 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, that is, the link is just crossing over the IG. 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:[4] S (Subject), O (Object), M (Modifier, adv/adj), P (Possessor), C (Classifier), D (Determiner), T (Dative Adjunct), L (Locative Adjunct), A: (Ablative Adjunct) and I (Instrumental Adjunct). For instance, with three channels, the two IGs of *bahçedeki* in Figure 2, would be represented as <MDO(bahçe+Noun+A3sg+Pnon+Loc)000> <000(+Det@)00d>. The M and the D to the left of

[1] Literally, "(the thing existing) at the time we caused (something) to become strong". Obviously this is not a word that one would use everyday. Turkish words found in typical text average about 3-4 morphemes including the stem.

[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.

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

[4] 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.



Last line shows the final POS for each word.

Figure 2: Dependency Links in an example Turkish Sentence

the first IG indicate the incoming modifier and determiner links, and the d on the right of the second IG indicates the outgoing determiner link.

4.2 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,
3. we remove the temporary brackets.

A typical linking rule looks like the following:[5]

```
[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), skipping over 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 word-final IG (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 any of the lower channels (the only symbols on the right side are 0s and 1s.)

For instance the example rule

```
[LL NominativeNominalA3pl LR] [ML AnyIG MR]*
[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** 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 bracketing rule defined as follows:

```
Bracket = [
          Pattern1 (->) "{Rel1" ... "Rel1}",
          Pattern2 (->) "{Rel2" ... "Rel2}",
          . . .
          ];
```

which will produce all possible bracketing of the input IG sequence.[6]

4.3 Filtering Crossing Link Configurations

The bracketings produced by **Bracket** contain configurations that may have 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:

```
(i) {S < ... 1 ... 0 ( ... ) ...
(ii)         ... ( ... ) 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 channel, e.g., the corresponding segment one of the previous channels may be all 0s. This constraint filters such cases to

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

[6] {Rel*i* and Rel*i*} are pairs of brackets; there is a distinct pair for each syntactic relation to be identified by these rules.

prevent redundant configurations from proliferating for later iterations of the parser.[7] For these two configuration constraints we define **FilterConfigs** as[8]

```
FilterConfigs = [ FilterCrossingLinks .o.
                  FilterEmptySegments];
```

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

```
Phase = Bracket  .o. FilterConfigs .o.
        MarkChannels .o. RemoveTempBrackets;
```

The transducer **MarkChannels** modifies the channel symbols in the bracketed segments to either the syntactic relation start or end symbol, or a 1, depending on the IG. Finally, the transducer **RemoveTempBrackets**, removes the brackets.[9]

The formulation up to now 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.,

```
Parse = Phase .o. Phase;
```

4.4 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 a finite state filter 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:[10]

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

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.[11] Many other syntactic constraints (e.g., only one object to a verb) can be formulated similar to above. All such constraints **Cons1, Cons2 ...ConsN**, can then be composed to give one FST that enforces all of these:

```
SyntacticFilter = [ Cons1 .o. Cons2 .o.
                    Cons3 .o.   ...   .o. ConsN]
```

4.5 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 =
    ["<" LeftChannelSymbols+
        "(" AnyIG ("@") ")"
         RightChannelSymbols+ ">"]* -
    ["<" LeftChannelSymbols*  0
        "(" AnyIG ("@") ")"
       0  RightChannelSymbols* ">"]*;
```

be a transducer which detects if any configuration has at least one link established in the last channel added (i.e., not all of the "topmost" channel symbols are 0's.) Let **MorphologicalDisambiguator** be 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 sentence to a transducer representing
  a chart of IGs
  M = [Sentence .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 empty.)

The step after the loop, **M = M .o. OnlyOneUnlinked**, enforces the constraint that

[7] 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.

[8] .o. denotes the transducer composition operator. We also use, for exposition purposes, =, instead of the XRCE define command.

[9] The details of these regular expressions are quite uninteresting.

[10] LeftChannelSymbols and RightChannelSymbols denote the sets of symbols that can appear on the left and right side channels.

[11] The crucial portion at the beginning says "For any IG it is not the case that there is more than one substring containing D among the left channel symbols of that IG."

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.

4.6 Robust Parsing

It is possible that either because of grammar coverage, or ungrammatical input, a parse with only one unlinked word final IG may not be found. In such cases `Parses` above would be empty. One may however opt to accept parses with $k > 1$ unlinked word final IGs when there are no parses with $< k$ unlinked word final IGs (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.

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  ];
```

which rejects configurations having more than 2 word final IGs whose right channel symbols contain only `0`s and `1`s, i.e., they do not link to some other IG as a dependent.

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 word final IGs, when there are no outputs with a smaller number of unlinked word final IGs. 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.

5 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. We have tested the resulting system and grammar on a corpus of 50 Turkish sentences, 20 of which were also used for developing and testing the grammar. These sentences had 4 to 24 words with an average 10 about 12 words.

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 (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 morphological disambiguation) is generated. 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 has 2707 states 27,713 transitions while the `SyntacticConstraints` transducer has 28,894 states and 302,354 transitions. The combined transducer for morphological analysis and (very limited) disambiguation has 87,475 states and 218,082 arcs.

Table 1 presents our results for parsing this set of 50 sentences. The number of iterations also count the last iteration where no new links are added. Inspired by Lin's notion of structural complexity (Lin, 1996), measured by the total length of the links in a dependency parse, we ordered the parses of a sentence using this measure. In 32 out of 50 sentences (64%), the correct parse was either the top ranked parse or among the top ranked parses with the same measure. In 13 out of 50 parses (26%) the correct parse was not among the top ranked parses, but was ranked lower. Since smaller structural complexity requires, for example, verbal adjuncts, etc. to attach to the nearest verb wherever possible, topicalization of such items which brings them to the beginning of the sentence, will generate a long(er) link to the verb (at the end) increasing complexity. In 5 out of 50 sentences (5%), the correct parse was not available among the parses generated, mainly due to grammar coverage. The parses generated in these cases used other (morphological) ambiguities of certain lexical items to arrive at some parse within the confines of the grammar.

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 (and manually) morphologically disambiguated input, parsing is instantaneous. Figure 4 presents the input and the output of the parser for a sample Turkish sentence. Figure 5 shows the output

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

English: World 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@)110><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@)110><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><00S(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 ***).

Figure 4: Sample Input and Output of the parser

Avg. Words/Sentence:	11.7 (4 – 24)
Avg. IGs/Sentence:	16.4 (5 – 36)
Avg. Parser Iterations:	5.2 (3 – 8)
Avg. Parses/Sentence:	23.9 (1 – 132)

Table 1: Statistics from Parsing 50 Turkish Sentences

of the parser processed with a Perl script to provide a more human-consumable presentation:

6 Discussion and Conclusions

We have presented the architecture and implementation of novel extended finite state dependency parser, with 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 (discontinuous) 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 (with backgrounding of some dependents of the main verb) can similarly be handled with a right-to-left version of `Parser` which is applied during each iteration, but these cases are very rare.

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 for immediately 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 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.

7 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.

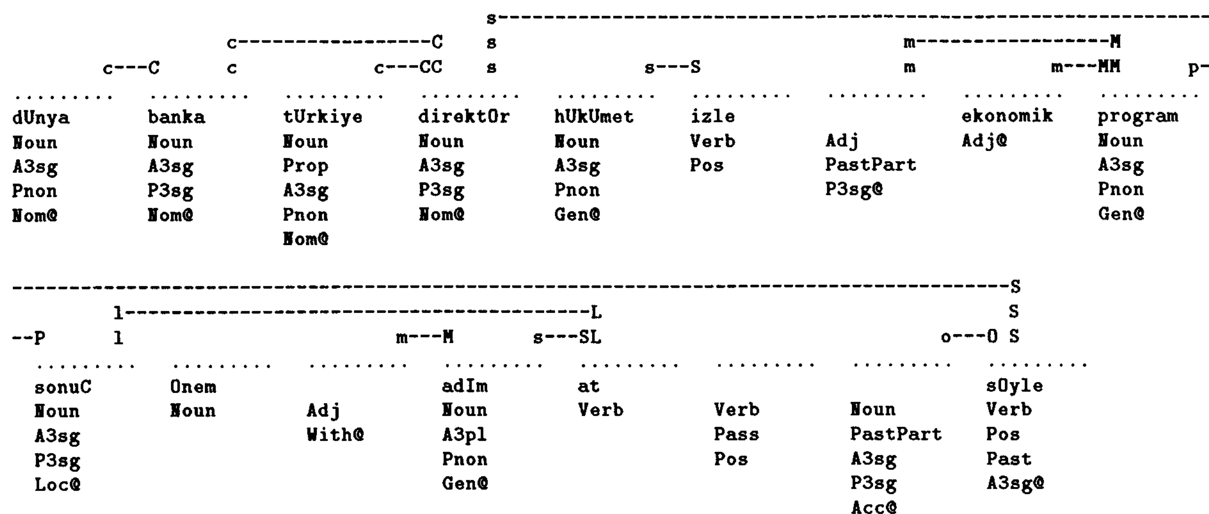


Figure 5: Dependency tree for the second parse

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.*

Dekang Lin. 1996. On the structural complexity of natural language sentences. In *Proceedings of the 16th International Conference on Computational Linguistics (COLING-96).*

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.

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.