Dependency Parsing with an Extended Finite-State Approach

Kemal Oflazer*
Sabancı University

This article 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 and no independent items except sentential head are filtered via finite-state filters. We have applied the parser to dependency parsing of Turkish.

1. Introduction

Finite-state machines have been used for many tasks in language processing, such as tokenization, morphological analysis, and parsing. 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, Pereira, and Riley 1998], and Finite State Automata Utilities [van Noord 1997]) 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, Tapanainen, and Voutilainen 1992; Grefenstette 1996; Chanod and Tapanainen 1996; Ait-Mokhtar and Chanod 1997; Hobbs et al. 1997). Another stream of work in using finite-state methods in parsing is based on approximating context-free grammars with finite-state grammars, which are then processed by efficient methods for such grammars (Black 1989; Pereira and Wright 1997; Grimley-Evans 1997; Johnson 1998; Nederhof 1998, 2000). 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 termination condition is reached (Abney 1996; Roche 1997).

This article presents an approach to dependency parsing using a finite-state approach. The approach is similar to those of Roche and Abney in that all three use an extended finite-state scheme to parse the input sentences. Our contributions can be summarized as follows:

- Our approach differs from Roche's and Abney's in that it is based on the dependency grammar approach and at the output produces an encoding of the dependency structure of a sentence. The lexical items and the dependency relations are encoded in an intertwined manner and manipulated by grammar rules, as well as structural and linguistic

* Faculty of Engineering and Natural Sciences, Sabancı University, Orhanlı, 34956, Tuzla, Istanbul, Turkey. E-mail: oflazer@sabanciuniv.edu.

constraints implemented as finite-state filters, to arrive at parses. The output of the parser is a finite-state transducer that compactly packs all the ambiguities as a lattice.

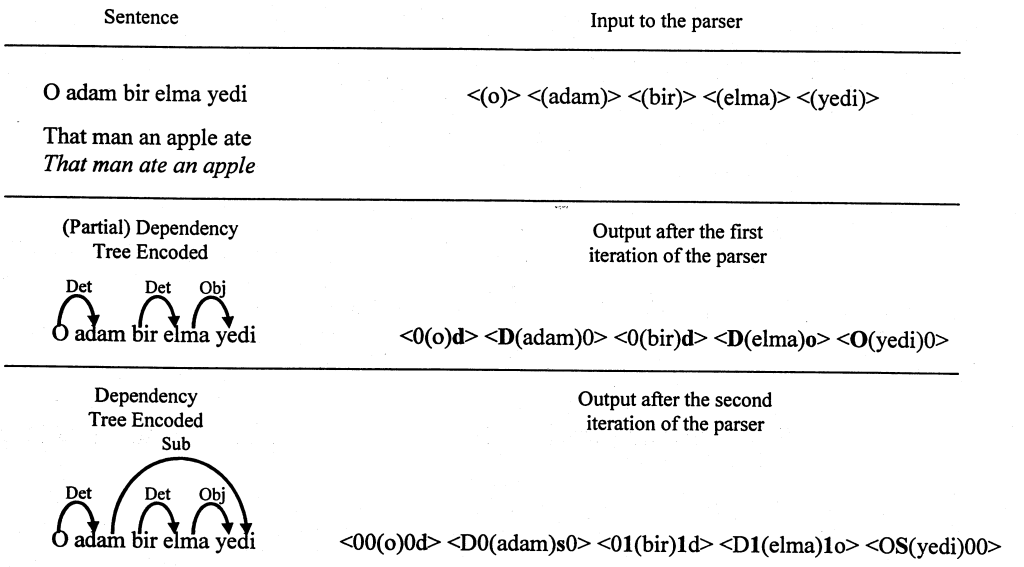
- As our approach is an all-parses approach with no statistical component, we have used Lin's (1995) proposal for ranking the parses based on the total link length and have obtained promising results. For over 48% of the sentences, the correct parse was among the dependency trees with the smallest total link length.
- Our approach can employ violable constraints for robust parsing so that when the parser fails to link all dependents to a head, one can use lenient filtering to allow parses with a small number of unlinked dependents to be output.
- The rules for linking dependents to heads can specify constraints on the intervening material between them, so that, for instance, certain links may be prevented from crossing **barriers** such as punctuation or lexical items with certain parts of speech or morphological properties (Collins 1996; Giguet and Vergne 1997; Tapanainen and Järvinen 1997).

We summarize in Figure 1 the basic idea of our approach. This figure presents in a rather high-level fashion, for a Turkish and an English sentence, the input and output representation for the approach to be presented. For the purposes of this summary, we assume that none of the words in the sentences have any morphological ambiguity and that their morphological properties are essentially obvious from the glosses. We represent the input to the parser as a string of symbols encoding the words with some additional delimiter markers. Panel (a) of Figure 1 shows this input representation for a Turkish sentence, on the top right, and panel (b) shows it for an English sentence.

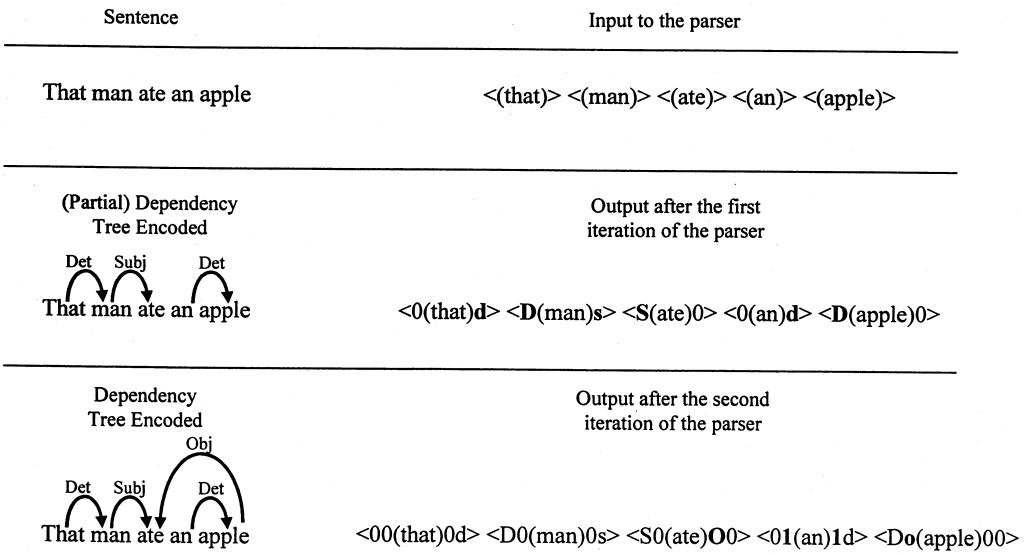
The parser operates in iterations. In the first iteration, the parser takes the input string encoding the sentence and manipulates it to produce the intermediate string in which we have three dependency relations encoded by additional symbols (highlighted with boldface type) injected into the string. The partial dependency trees encoded are depicted to the left of the intermediate strings. It should be noted that the sets of dependency relations captured in the first iteration are different for Turkish and English. In the Turkish sentence, two determiner links and one object link are encoded in parallel, whereas in the English sentence, two determiner links and one subject link are encoded in parallel. The common property of these links is that they do not "interfere" with each other.

The second iteration of the parser takes the output of the first iteration and manipulates it to produce a slightly longer string in which symbols encoding a new subject (object) link are injected into the Turkish (English) string. (We again highlight these symbols with boldface type.) Note that in the English string the relative positions of the link start and end symbols indicate that this is a right-to-left link. The dependency structures encoded by these strings are again on their left. After the second iteration, there are no further links that can be added, since in each case there is only one word left without any outgoing links and it happens to be the head of the sentence.

The article is structured as follows: After a brief overview of related work, we summarize dependency grammars and aspects of Turkish relevant to this work. We provide a summary of concepts from finite-state transducers so that subsequent sections can be self-contained. We continue by describing the representation that we have employed for encoding dependency structures, along with the encoding of dependency linking rules operating on these representations and configurational constraints



(a) Dependency parsing for a Turkish sentence



(b) Dependency parsing for an English sentence

Det: Determiner Link(d/D), Obj: Object Link (o/O), Subj: Subject Link (s/S)

Figure 1
Dependency parsing by means of iterative manipulations of strings encoding dependency structures.

for filtering them. We then describe the parser and its operational aspects, with details on how linguistically motivated constraints for further filtering are implemented. We briefly provide a scheme for a robust-parsing extension of our approach using the lenient composition operation. We then provide results from a prototype implementation of the parser and its application to dependency parsing of Turkish. We close with remarks and conclusions.

2. Overview of Related Work

Although finite-state methods have been applied to parsing by many researchers, extended finite-state techniques were initially used only by Roche (1997), Abney (1996), and the FASTUS group (Hobbs et al. 1997). In the context of dependency parsing with finite-state machines, Elworthy (2000) has recently proposed a finite-state parser that produces a dependency output.

Roche (1997) presents a top-down approach for parsing context-free grammars implemented with finite-state transducers. The transducers are based on a syntactic dictionary comprising patterns of lexical and nonlexical items. The input is initially bracketed with sentence markers at both ends and then fed into a transducer for bracketing according to bracketing rules for each of the patterns in the dictionary. The output of the transducer is fed back to the input, and the constituent structure is iteratively refined. When the output of the transducer reaches a fixed point, that is, when no additional brackets can be inserted, parsing ends.

Abney (1996) presents a finite-state parsing approach in which a tagged sentence is parsed by transducers that progressively transform the input into sequences of symbols representing phrasal constituents. In this approach, the input sentence is assumed to be tagged with a part-of-speech tagger. The parser consists of a cascade of stages. Each stage is a finite-state transducer implementing rules that bracket and transduce the input to an output containing a mixture of unconsumed terminal symbols and nonterminal symbols for the phrases recognized. The output of a stage goes to the next stage in the cascade, which further brackets the input using yet other rules. Each cascade typically corresponds to a level in a standard X-bar grammar. After a certain (fixed) number of cascades, the input is fully bracketed, with the structures being indicated by labels on the brackets. Iterations in Roche's approach roughly correspond to cascades in Abney's approach. The grammar, however, determines the number of levels or cascades in Abney's approach: that is, structure is fixed. A work along the lines of Abney's is that of Kokkinakis and Kokkinakis (1999) for parsing Swedish.

Elworthy (2000) presents a finite-state parser that can produce a dependency structure from the input. The parser utilizes standard phrase structure rules that are annotated with "instructions" that associate the components of the phrases recognized with dependency grammar–motivated relations. A head is annotated with variables associating it with its dependents. These variables are filled in by the instructions associated with the rules. These variables are copied or percolated "up" the rules according to special instructions. The approach resembles a unification-based grammar in which instead of unification, dependency relation features are passed from a dependent to its head. The rules for recognizing phrases and implementing their instructions are implemented as finite-state transducers.

Another notable system for finite-state parsing is FASTUS (Hobbs et al. 1997). FASTUS uses a five-stage cascaded system, with each stage consisting of nondeterministic finite-state machines. FASTUS is mainly for information extraction applications. The early stages recognize complex multiword units such as proper names and colloca-

tions and build upon these by grouping them into phrases. Later stages are geared toward recognizing event patterns and building event structures.

3. Dependency Syntax

Dependency approaches to syntactic representation use the notion of syntactic relation to associate surface lexical items. 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 was held at COLING/ACL'98). Järvinen and Tapanainen (1998; Tapanainen and Järvinen 1997) have demonstrated an efficient wide-coverage dependency parser for English. The work of Sleator and Temperley (1991) on link grammar, essentially a lexicalized variant of dependency grammar, has also proved to be interesting in regard to a number of aspects. Dependency-based statistical language modeling and parsing have also become quite popular in statistical natural language processing (Lafferty, Sleator, and Temperley 1992; Eisner 1996; Chelba et al. 1997; Collins 1996; Collins et al. 1999).

Robinson (1970) gives four axioms for well-formed dependency structures that have been assumed in almost all computational approaches. These state that, in a dependency structure of a sentence,

1. one and only one word is independent, that is, not linked to some other word;
2. all others depend directly on some word;
3. no word depends on more than one other; and
4. if a word *A* depends directly on word *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., Lai 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).[1]

4. Turkish

Turkish is an agglutinative language in which a sequence of inflectional and derivational morphemes get affixed to a root (Oflazer 1993). At the syntax level, the unmarked constituent order is Subject-Object-Verb, but constituent order may vary as demanded by the discourse context. Essentially all constituent orders are possible, especially at the main sentence level, with very minimal formal constraints. In written text, however, the unmarked order is dominant at both the main-sentence and embedded-clause level.

Turkish morphophonology is characterized by a number of processes such as vowel harmony (vowels in suffixes, with very minor exceptions, agree with previous

1 See section 6 for how projectivity is checked and section 6.5 on the implications of checking for projectivity during parsing with both right-to-left and left-to-right dependency links.

vowels in certain aspects), consonant agreement, and vowel and consonant ellipsis. The morphotactics are quite complicated: A given word form may involve multiple derivations (as we show shortly). The number of word forms one can generate from a nominal or verbal root is theoretically infinite (see, e.g., Hankamer, [1989]).

Derivations in Turkish 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 (possibly intermediate) derived forms. In this work, we assume that a Turkish word is represented as a sequence of **inflectional groups** (IGs), separated by ^DBs, denoting derivation boundaries, in the following general form:

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

Here each $\texttt{IG}_i$ denotes relevant inflectional features including the part of speech for the root, for the first IG, and for any of the derived forms. For instance, the derived modifier sağlamlaştırdığımızdaki[2] would be represented as[3]

```
sağlam+Adj+^DB+Verb+Become+^DB+Verb+Caus+Pos
              +^DB+Noun+PastPart+A3sg+P3sg+Loc
              +^DB+Adj
```

The five IGs in this are

1. sağlam(strong)+Adj
2. +Verb+Become
3. +Verb+Caus+Pos
4. +Noun+PastPart+A3sg+P3sg+Loc
5. +Adj

The first shows the root word along with its part of speech, which is its only inflectional feature. The second IG indicates a derivation into a verb whose semantics is "to become" the preceding adjective. The +Become can be thought of as a minor part-of-speech tag. The third IG indicates that a causative verb with positive polarity is derived from the previous verb. The fourth IG indicates the derivation of a nominal form, a past participle, with +Noun as the part of speech and +PastPart. It has has other inflectional features: +A3sg for third-person singular, +P3sg for third-person singular possessive agreement, and +Loc for locative case. Finally the fifth IG indicates a derivation into an adjective.

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 emanate only from the last IG of a (dependent) word and land on one of the IGs of a (head)word on the

2 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 three to four morphemes including the stem, with an average of about 1.7 derivations per word.

3 The morphological features other than the obvious part-of-speech features are +Become: become verb, +Caus: causative verb, PastPart: derived past participle, P3sg: third-person singular possessive agreement, A3sg: third-person singular number-person agreement, +Zero: zero derivation with no overt morpheme, +Pnon: no possessive agreement, +Loc: locative case, +Pos: positive polarity.

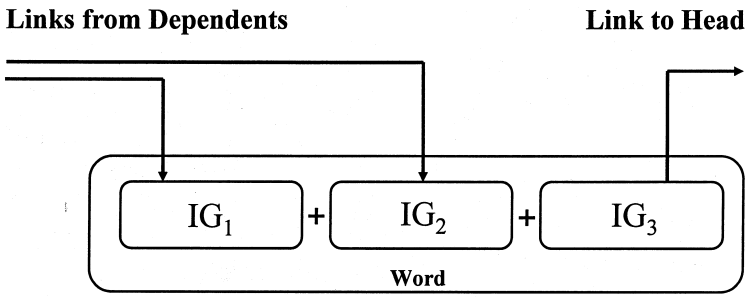


Figure 2
Links and inflectional groups.

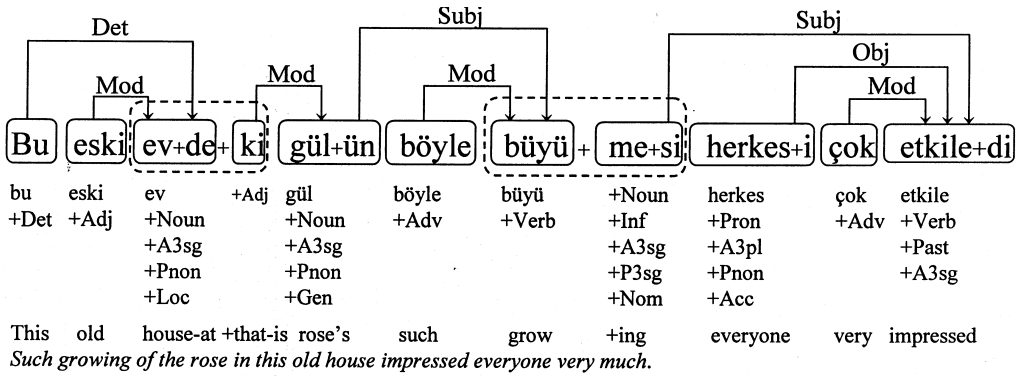


Figure 3
Dependency links in an example Turkish sentence.

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.[4] Figure 3 shows a dependency tree for a Turkish sentence laid on top of the words segmented along IG boundaries. It should be noted that all IGs that link to the same head IG comprise a constituent, and the legality of a link depends primarily on the inflectional features of the IGs it connects.

For the purposes of this article we can summarize aspects of Turkish as follows:

- The IGs are the "words." That is, we treat a chunk of (free and bound) inflectional morphemes as the units that we relate with dependency links.
- Within a word, the IGs are linearly dependent on the next IG, if any. We would not, however, show and deal with these explicitly, but rather deal only with the dependency link emanating from the last IG in each word, which is the syntactic head of the word (whereas the first IG which contains the root is the semantic head).

4 Such cases would be violating the projectivity constraint. The only examples of such crossing that we know are certain discontinuous noun phrases in which an adverbial modifier of the matrix verb intervenes between a specifier and the rest of the noun phrase. Since the specifier links to the head noun but the adverbial links to the verb, the links have to cross.

- For all practical purposes the syntactic dependency links go from left to right, that is, the core structure is subject-object-verb, and modifiers precede their heads.

5. Finite-State Transducers

The exposition in the subsequent sections will make extensive use of concepts from finite-state transducers. In this section, we provide a brief overview of the main relevant concepts; the reader is referred to recent expositions (e.g., Roche and Schabes [1997]; also, Hopcroft and Ullman [1979] provides a detailed exposition of finite-state machines and regular languages.)

Finite-state transducers are finite-state devices with transitions labeled by pairs of symbols `(u:l)`, `u` denoting the "upper" symbol and `l` denoting the "lower" symbol. These symbols come from a finite alphabet. Additionally, either `u` or `l` (but not both) can be the ϵ symbol, denoting the empty string. A finite-state transducer T maps between two regular languages: U, the "upper" language, and L, the "lower" language. The mapping is bidirectional, and in general, a string in one of the languages may map to one or more strings in the other language. The transductions in both directions are valid only if the string on the input side takes the finite-state transducer to a final state.

The behavior of finite-state transducers can also be described using regular expressions over an alphabet of symbols of the form `(u:l)` (including symbols ϵ`:l` and `u:`ϵ), in complete analogy to regular expressions for finite-state recognizers. Since the notational mechanisms provided by the basic definition of regular expressions (concatenation, union, and Kleene star [Hopcroft and Ullman 1979]) are quite restricted and low level, developers of finite-state transducer manipulation systems have augmented the notational capabilities with operations at a much higher level of abstraction, much closer to the operations used by the computational linguistics application (see, e.g., Karttunen et al., [1996]; see also http://www.xrce.xerox.com/competencies/content-analysis/fsCompiler/fssyntax.html, and also van Noord, [1997]).

Finite-state transducers are closed under union, but in contrast to finite-state recognizers, they are not closed under difference and intersection operations (Kaplan and Kay 1994). On the other hand, finite-state transducers are closed under the operation of **composition**, which is very much an analog of function composition in algebra. Let T_1 be a transducer that maps between regular languages U_1 and L_1, and let T_2 be a transducer that maps between regular languages U_2 and L_2. The composition T of T_1 and T_2, denoted by $T_1 \circ T_2$, is the transducer that maps between $U = T_1^{-1}(L_1 \cap U_2)$ and $L = T_2(L_1 \cap U_2)$.[5] That is, the resulting mapping is defined only for the respective images, in T_1^{-1} and T_2, of the intersection $L_1 \cap U_2$. A pair of strings $(x, y) \in T_1 \circ T_2$ if and only if $\exists z$ such that $(x, z) \in T_1$ and $(z, y) \in T_2$. Note that the composition operation is order dependent; $T_1 \circ T_2$ is not the same mapping as $T_2 \circ T_1$. Figure 4 summarizes the main points of the composition operation for finite-state transducers.

6. Finite-State Dependency Parsing

Our approach is based on constructing a graphic representation of the dependency structure of a sentence, including the lexical items and the labeled directed arcs en-

5 Notationally, for a transducer T, we take $T(U)$ to mean the transduction from the upper to the lower language and T^{-1} to mean the transduction from the lower to the upper language.

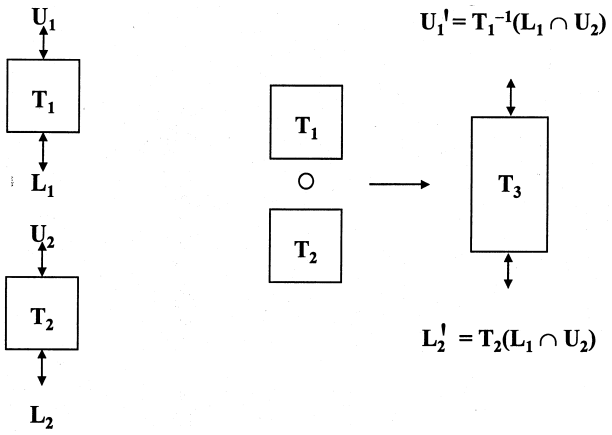


Figure 4
Composition operation for finite-state transducers.

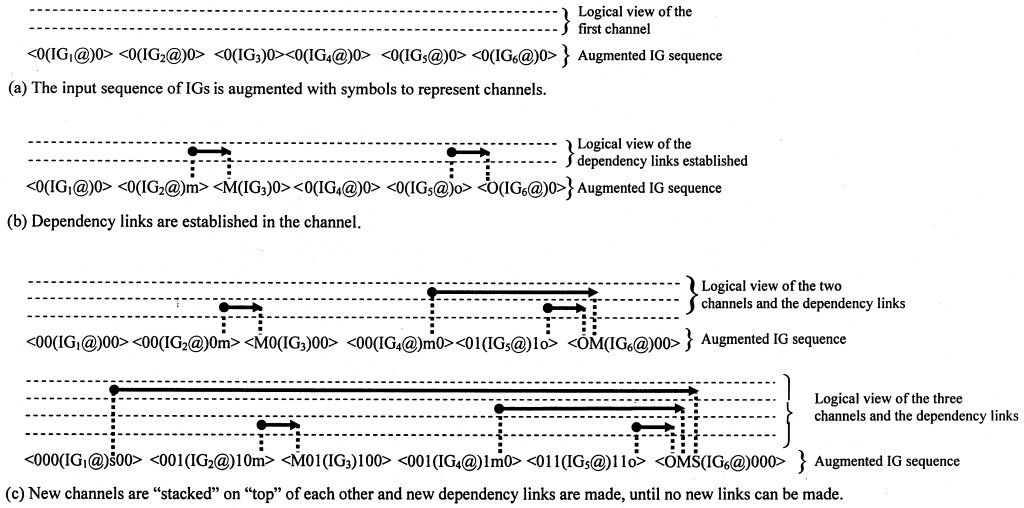


Figure 5
Physical representation and logical view of channels and dependency links.

coding the dependency relations. In constructing this dependency graph, these labeled links are represented by additional symbols that are laid out within the symbols representing the lexical items and their morphological features.

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 in Figure 5(a). The input to the parser is a representation of the sentence as a sequence (or a lattice, if morphological disambiguation has not been performed) of IGs with some additional symbols to delineate certain boundaries.[6] The

6 The lattice of all morphological analyses of the words in a sentence can be encoded by a finite-state acceptor. In fact, one gets such an acceptor when a nondeterministic morphological analysis finite-state transducer is applied to the input. Further, finite-state acceptors are assumed to be coerced into **identity transducers** that map the input strings they accept to identical output strings. This coercion is necessary so that filters defined as acceptors can be used as transducers with the composition operators.

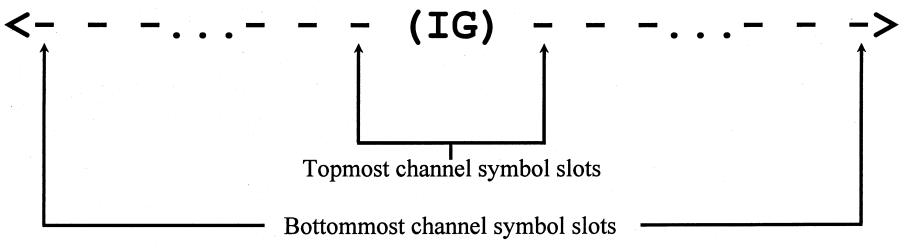


Figure 6
Channel symbol slots around an inflectional group.

parser operates in a number of iterations: At each iteration of the parser, a new empty channel is "stacked" on "top" of the input, and any possible links are established using these channels.[7] Parsing terminates when no new links can be established within the most recent channel added, that is, when a fixed point is reached. In this respect, this approach is similar to that of Roche (1997). An abstract view of this is presented in panels (a) through (c) of Figure 5.

6.1 Representing Channels and Syntactic Relations

The sequence (or the lattice) of IGs is produced by a morphological analysis transducer, with each IG initially being augmented by two pairs of delimiter symbols, as `<(IG)>`. The `(` and `)` pair separates the morphological features from the channel representation symbols, while `<` and `>` separate the representations of consecutive IGs. Word-final IGs (IGs from which links will emanate) are further augmented with a special marker `@`.

Channels are represented by pairs of matching symbols that are inserted between the `<...(` and the `)...>` delimiter symbols. Symbols for new channels (upper channels in Figure 5) are stacked so that the symbols for the topmost channels are those closest to the `(...)`, and in this way dependency links do not cross when drawn (see Figure 6). At any time, the number of channel symbols on both sides of an IG is the same. Multiple dependency links can occupy mutually exclusive segments of a channel as long as they do not interfere with each other; that is, each channel may accommodate many dependency links whenever possible.[8]

How a certain segment of channel is used is indicated by various symbols surrounding the IGs, within the `<` and `>` delimiters:

- The channel symbol `0` indicates that the channel segment is not used by any dependency link and thus is empty.
- The channel symbol `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" this IG.
- When a link starts from a word-final IG, then a link start symbol is used on the *right* side of the word-final IG (i.e., between `)` and `>`).
- When a link terminates on an IG, then a link end symbol denoting the syntactic relation is used on the *left* side of the IG (i.e., between `<` and `(`).

7 The beginnings and the ends of the arrows in the figure indicate the dependent and head IGs, respectively.
8 The exposition here is for only left-to-right links. See section 6.5 for a dependency representation for both left-to-right and right-to-left links.

The following syntactic relations are currently encoded in the channels:

1. Subject (`s/S`)
2. Object (`o/O`)
3. Modifier (adverbs/adjectives) (`m/M`))
4. Possessor (`p/P`)
5. Classifier (`c/C`)
6. Determiner (`d/D`)
7. Dative adjunct (`t/T`)
8. Ablative adjunct (`f/F`)
9. Locative adjunct (`l/L`)
10. Instrumental adjunct (`i/I`)

The lowercase symbol in each case is used to indicate the start of a link, and the uppercase symbols indicate the end of a link. Both kinds of symbols are used to encode configurational and linguistic constraints on IGs, as we show later.

For instance, with three channels, the dependency structure of the IGs of *bu eski evdeki gülün* (of the rose at this old house) in Figure 3 would be represented as

```
<000(bu+Det@)0d0><010(eski+Adj@)01m><MD0(ev+Noun+A3sg+Pnon+Loc)000>
<000(+Adj@)00m><M00(gül+Noun+A3sg+Pnon+Gen@)0p0>)
```

The `M` and the `D` to the left of the first IG of *evdeki* (third IG above) indicate the incoming modifier and determiner links from the first two IGs, matching the start symbols `m` and `d` in the second and the first IGs. The `m--M` pair encodes the modifier link from *eski* (old) to *evde* (at house), and the `d--D` pair encodes the determiner link from *bu* (this) to *evde*. The last IG above has an `M` on the left side matching the `m` in the IG to the left. This `m--M` pair encodes the modifier relation between *+ki* and *gülün* (of the rose). The last IG above has an outgoing possessor link marked by the `p` on its right side, indicating that it is a genitive-marked possessor of some other IG to the right.

We should note, however, that the (morphological) relations between IGs that make up a single word are not at all a concern here and are not considered to be syntactic dependency relations. Thus they are never explicitly shown or encoded except by virtue of their being sequentially placed in the input. The only links that we explicitly encode are those links emanating from a word-final IG and landing on some other IG.

6.2 Components of the Parser

The basic strategy of a parser iteration is to recognize, by means of a rule (encoded as a regular expression), a dependent IG and a head IG and link them by modifying the "topmost" channel between the two. Once we identify the dependent IG and the head IG (in a manner to be described shortly), we proceed as follows:

1. We create an empty channel by injecting **0**s to just outside of the `(...)` pairs:

<...**0**(ID$_{dep}$@)**0**...>...<...**0**(IG)**0**...>...<...**0**(IG$_{head}$)**0**...>

2. We put temporary braces (of the type for the dependency link to be established—we use MOD below for a modifier link for expository purposes) to the right of the dependent IG and to the left of the head IG:

<...0(IG$_{dep}${**MOD**@)0...>...<...0(IG)0...>...<...0**MOD**}(IG$_{head}$)0...>

3. We mark the start, intermediate, and end IGs of the link with the appropriate symbols encoding the relation thus established by the braces:

<...0(IG$_{dep}${MOD@)**m**...>...<...**1**(IG)**1**...>...<...**M**MOD}(IG$_{head}$)0...>

4. We remove the temporary braces, and we have the string in item 1, with some symbols modified:

<...0(IG$_{dep}$@)m...>...<...1(IG)1...>...<...M(IG$_{head}$)0...>

The second step above deserves some more attention, as it has additional functions besides identifying the relevant IGs. One should be careful to avoid generating strings that are either illegal or redundant or cannot lead to a valid parse at the end of the iterations. Thus, the second step makes sure that

1. The last channel in the segment to be bracketed is free.
2. The dependent is not already linked at one of the lower channels (since an IG can be the dependent of only one other IG).
3. None of the channels directly underneath the segment have any links coming into or going out of the projection, in those channels, of the segment bracketed. This makes sure that there are no crossing links. It is obviously okay to have links that start and terminate in lower channels within the projection of the bracketed segment.[9]
4. There are no channels below the current channel that are unused in the segment to be bracketed (if there are, then this link could have been made there.)
5. The link to be established does not trap an *unlinked* word-final IG. If there is such an IG, its future link would have to cross the link to be established in the current segment.

The last three of these constraints are depicted in Figure 7.

6.3 Rules for Establishing Dependency Links

The components of a dependency link are recognized using regular expressions. These regular expressions identify the dependent IG, the head IG, and the IGs in between to be skipped over, and they temporarily bracket the input segment including these IGs.

9 It is actually possible to place a crossing link by laying it in a special channel below the IG sequence so that it would not interfere with the other links. This would necessitate additional delimiter symbols and would unnecessarily further complicate the presentation.

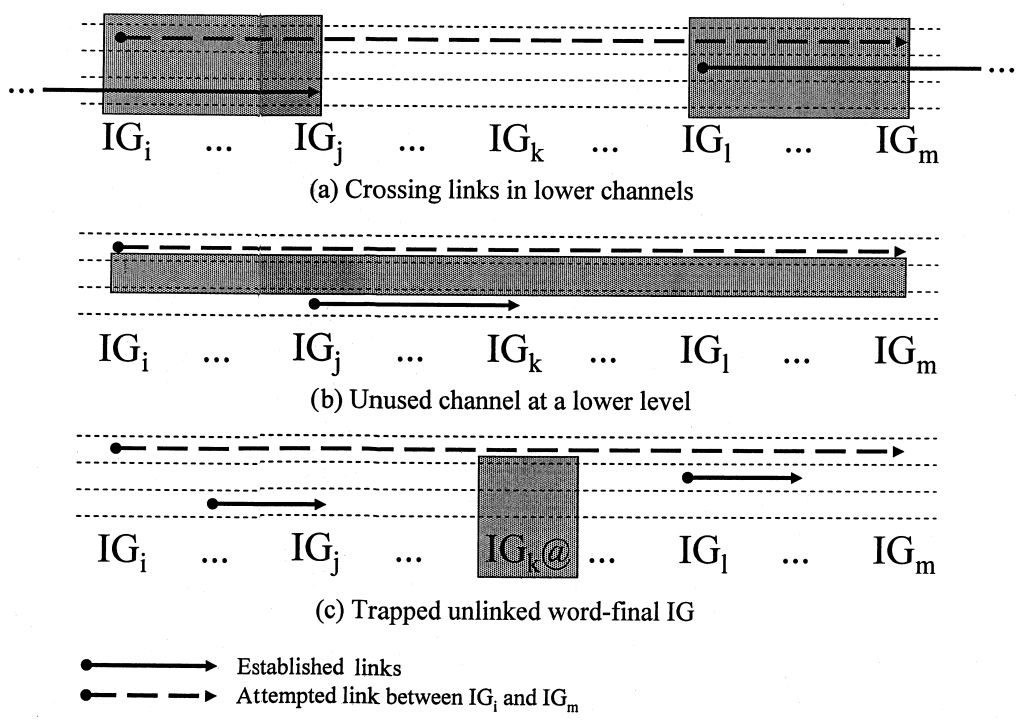


Figure 7
Configurations to be avoided during parsing.

These regular expressions correspond to "grammar rules," and a collection of these rules comprise the dependency grammar.

A typical rule looks like the following:[10]

```
[ LR [ML IGMiddle MR]* RL ] (->) "{Rel" ... "Rel}" || IGDep _ IGHead
```

This rule is an example of a XRCE optional-replace rule that nondeterministically inserts the curly braces on the right-hand side (the symbols on both sides of the ellipsis) into the output string in the lower language, around any part of the input string in the upper language that matches its left-hand side, provided the left-hand side is contextually constrained on the left by `IGDep` and on the right by `IGHead`. This replace rule can nondeterministically make multiple nonoverlapping replacements.[11]

The left-hand side of this rule (to the left of `(->)`) has three components: The first part, `LR`, specifies the constraints on the right-hand side of the dependent IG. The second part, `[ML IGMiddle MR]*`, defines *any* middle IGs that will be ignored

10 We use the XRCE regular expression language syntax; the `[` and `]` act like parentheses used as grouping operators in the language. See http://www.xrce.xerox.com/competencies/content-analysis/fsCompiler/fssyntax.html for details.

11 An earlier implementation of the parser used a slightly different optional-replace rule that did not make use of the contextual constraint, as the new format was not included in the toolkit available to the author. A typical rule there looked like `[[LL IGDep LR] [ML IGMiddle MR]* [RL IGHead RR]] (->) "{Rel" ... "Rel}"`. Although for the purposes of writing the dependency grammar, the old rule format was more transparent, its use necessitated some extra complexity in various other components of the parser. The old rule format has been abandoned in favor of the new format. I thank an anonymous reviewer for suggesting the use of this new rule format.

and skipped over,[12] and the third part, RL, specifies the contraints on the left-hand side channel symbols of the head IG. The head and dependent IG patterns IGDep and IGHead are specified as left and right contextual constraints on the pattern of the three components specified on the left-hand side.

This rule (optionally) brackets (with {*Rel* and *Rel*}) any occurrence of pattern LR [ML IGMiddle MR] RL provided the pattern IGDep is to the left of LR and the pattern IGHead is to the right of RL.[13] After the (optional) bracketing, the brace {*Rel* occurs between IGDep and LR, and the brace *Rel*} occurs between RL and IGHead. Each rule has its own brace symbol depending on the relationship of the dependent and the head. The optionality is necessary because a given IG may be related to multiple IGs as a result of syntactic ambiguities, and all such links have to be produced to arrive at the final set of parses. It should also be noted that there are rules that deviate from the template above in that the segment to be skipped may be missing, or may contain barrier patterns that should not be skipped over, etc.

The symbols L(eft)R(ight), M(iddle)L, MR, and RL are regular expressions that encode constraints on the bounding channel symbols that are used to enforce some of the configurational constraints described earlier. Let

```
RightChannelSymbols = [  "1" | "0" | "s" | "o" | "m" | "p" |
                         "c" | "d" | "t" | "l" | "f" | "i"  ];
```

and

```
LeftChannelSymbols =  [  "1" | "0" | "S" | "O" | "M" | "P" |
                         "C" | "D" | "T" | "L" | "F" | "I" ];
```

These four regular expressions are defined as follows:

1. The regular expression LR = ["@" ")" "0" ["0"]* ">"] checks that
   a. The matching IG is a word-final IG (has a @ marker)
   b. The right-side topmost channel is empty (channel symbol nearest to) is 0)
   c. The IG is not linked to any other in any of the lower channels
   d. No links in any of the lower channels cross into this segment (that is, there are no 1s in lower channels.)

   These conditions imply that the only channel symbol that may appear in the right side of a dependent IG is 0.
2. The regular expression ML = ["<" LeftChannelSymbols* "0" "("] ensures that the topmost channel is empty, but it does not constrain the symbols in the lower channels, if any, as there may be other links ending at the matching IG.
3. Similarly, the regular expression MR = [")" "0" RightChannelSymbols* ">"] also ensures that the topmost channel is empty, but it does not constrain the symbols in the lower channels, if any, as there may be other links starting at the matching IG.

12 It is possible that the pattern to be skipped over can be specified by more complex patterns.
13 We use the symbols {*Rel* and *Rel*} as generic bracketing delimiters.

4. The regular expression
   `RL =[ "<" [LeftChannelSymbols* - $1] "0" "(" ]` also ensures that the topmost channel is empty. Note that since the matching IG is the IG of the head, multiple dependency links may end at the matching IG, so there are no constraints on the symbols in the lower channels, but there cannot be any `1`s on the left side, since that would imply a lower link crossing to the right side.

For instance, the rule

```
[ LR [ ML AnyIG MR ]*  RL ]  (->) "{SBJ" ... "SBJ}" ||
                              NominativeNominalA3pl _ FiniteVerbA3sgA3pl;
```

is used to bracket a segment starting right after a plural nominative nominal, as subject of a finite verb somewhere on the right, with either `+A3sg` or `+A3pl` number-person agreement (allowed in Turkish). In this rule, the regular expression `Nominative-NominalA3pl` is defined as follows:

```
 [ (RootWord) ["+Noun" |"+Pron"] (NominalType) "+A3pl"
                                                PossessiveAgreement "+Nom"]
```

and it matches any nominal IG (including any derived nominals) with nominative case and `+A3pl` agreement. There are a number of points to note in this expression:

1. `(...)` indicates optionality: The `RootWord`, a regular expression matching a sequence of one or more characters in the Turkish alphabet, is optional, since this may be a derived noun for which the root would be in a previous IG.
2. `NominalType`, another optional component, is a regular expression matching possible minor part-of-speech tags for nouns and pronouns.
3. `PossessiveAgreement` is a regular expression that matches all possible possessive agreement markers.
4. The nominal has third-person plural agreement and nominative case.

The order of the components of this regular expression corresponds to the order of the morphological feature symbols produced by the morphological analyzer for a nominal IG. The regular expression `FiniteVerbA3sgA3pl` matches any finite-verb IG with either `+A3sg` or `+A3pl` number-person agreement. The regular expression `AnyIG` matches any IG.

All the rules in the dependency grammar written in the form described are grouped together into a parallel bracketing regular expression defined as follows:

```
Bracket = [
             [LR [ML IGMiddle1 MR]* RL] (->) "{Rel1" ... "Rel1}"
                                                || IGDep1 _ IGHead1,,
             [LR [ML IGMiddle2 MR]* RL] (->) "{Rel2" ... "Rel2}"
                                                || IGDep2 _ IGHead2,,
          . . .
             [LR [ML IGMiddlen MR]* RL] (->) "{Reln" ... "Reln}"
                                                || IGDepn _ IGHeadn
          ];
```

where left-hand-side patterns and dependent and head IGs are specified in accordance with the rule format given earlier. $\{Rel_i$ and $Rel_i\}$ are pairs of braces; there is a distinct pair for each syntactic relation to be identified by these rules (and not necessarily a unique one for each rule). This set of rules will produce all possible bracketings of the input IG sequence, subject to the constraints specified by the patterns. This overgeneration is then filtered (mostly at compile time and not at parse time) by a number of additional configurational and linguistic constraints that are discussed shortly.

6.4 Constructing the Parsing Transducer

In this section, we describe the components of the parsing transducer. As stated earlier, links are established in a number of iterations. Each iteration mainly consists of an application of a parsing transducer followed by a filtering transducer that eliminates certain redundant partial parse configurations.[14]

The parsing transducer consists of a transducer that inserts an empty channel followed by transducers that implement steps 2 to 4 described at the beginning of section 6.2.

We can write the following regular expression for the parser transducer as[15]

```
Parser = AddChannel
           .o.
         Bracket
           .o.
       FilterEmptySegments
           .o.
         MarkChannels
           .o.
         RemoveBraces;
```

The transducer `AddChannel` is a simple transducer that adds a pair of `0` channel symbols around the `(...)` in the IGs. It implements step 1 in section 6.2. The transducer `Bracket` was defined in the previous section. It implements step 2 described in section 6.2.

Since the bracketing rules are nondeterministic, they will generate many configurations in which certain segments in the stacked channels will not be used. A rule may attempt to establish a link in the topmost channel even though the corresponding segment is not utilized in a previous channel (e.g., the corresponding segment of one of the previous channels may be all `0`s). One needs to eliminate such redundant configurations after each iteration to prevent their proliferation at later iterations of the parser. Checking whether the segment just underneath the topmost channel is empty has worked perfectly in our experiments in that none of the parses selected had any empty segments that were not detected by this test. The regular expression `FilterEmptySegments` filters these configurations, an example of which is depicted in Figure 7(b).[16]

14 We use the term **configuration** to denote an encoding of a (partial) dependency parse as a string of symbols, as described in section 6.1.

15 The `.o.` operator denotes the composition operation (denoted using ∘ in section 5) for finite-state transducers in the XRCE regular expression language.

16 Incorporating this configurational constraint into the bracketing phase implies that each rule will have a check encoding the fact "it is not the case that all symbols in the channel immediately below are `0`," which makes all rules a bit awkward. Incorporating this as a separate postbracketing constraint is simpler. Since all compositions are done at compile time, no additional penalty is incurred.

The transducer `MarkChannels` implements step 3 in section 6.2. This transducer modifies the channel symbols to mark a link:

- The new (topmost) right channel symbol in the IG just to the right of the opening brace is modified to a link start symbol (one of the symbols `s, o, m, p, c, d, t, l, f, i`).
- The new (topmost) right channel symbols on both sides of all the IGs fully bracketed by the braces (all IGs except the dependent and head IGs) are modified to `1`; this is necessary so that that segment of the channel can be claimed and used for detecting crossing links.
- The new (topmost) left channel symbol in the IG just to the left of the closing brace is modified to a link end symbol (one of the symbols `S, O, M, P, C, D, T, L, F, I`).

Finally, the transducer `RemoveBraces` removes the braces.[17] It should be noted that the transducer for `Parse` is computed off-line at compile time, so that no composition is done at parse time.

The parsing scheme described above is bottom up, in that the links between closest head-dependent pairs are established first, in the lowest channel. Subsequent longer-distance links are established in later stages as long as they do not conflict with links established in lower channels. It is also conceivable that one could employ a top-down parsing scheme linking all pairs that are far apart, again checking for configurational constraints. If full nondeterminism is maintained in the bracketing step, it really does not matter whether one uses bottom-up or top-down parsing. Bottom-up parsing, however, offers certain advantages in that various (usually linguistically motivated) constraints that have to hold between nearby pairs or pairs that have to be immediately sequential can be enforced at an earlier stage by using simpler regular expressions. These constraints help prune the intermediate parse strings.

6.5 Dependency Structures with Both Left-to-Right and Right-to-Left Links

Although the formulation up until now has been one for dependency structures involving left-to-right dependency links, the approach presented above can handle a dependency grammar with both left-to-right and right-to-left links. In this section, we will outline the details of the changes that would be needed in such a formulation but will then go ahead with the left-to-right implementation, as that forms the basis of our implementation for Turkish, for which left-to-right links suffice for all practical purposes.

Incorporating the right-to-left links into the approach would require the following modifications to the formulation:

1. The right-to-left links would use the same representation as the left-to-right links, except they would be distinguished by the symbols marking the links at the dependent and head IG sites. With the left-to-right links described so far, lowercase link symbols on the right side of an IG mark the dependent IG and uppercase symbols on the left

17 The details of the regular expressions for these transducers are rather uninteresting. They are essentially upper-side to lower-side contextual-replace regular expressions. For instance, `RemoveBraces` maps all brace symbols on the upper side to ϵ on the lower side.

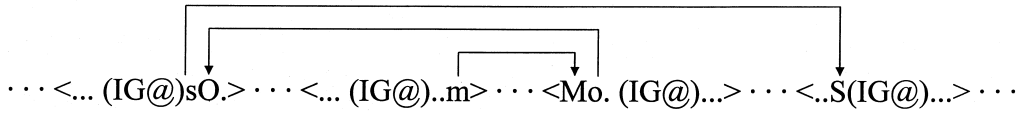


Figure 8
Bidirectional dependency links.

side of the IG mark the head IG (which follows the dependent IG in linear order). We would still use the same conventions for right-to-left links, except that we could have head and dependent IG markers on both sides of the channel representation. This is shown graphically in Figure 8. So a right-to-left link would have a lowercase link mark on the left side of the dependent IG and an uppercase link mark on the right side of the head IG to the left of the dependent IG.

2. With both left-to-right and right-to-left rules, we would need two different rule formats. The rule format for left-to-right links would be slightly different from the format given earlier:

```
[ LRl [ML IGMiddle MR]* RLl ] (->) "{Rel-left-to-right"
                                   ... "Rel-left-to-right}"
                                        || LL IGDep _ IGHead The
```

rule format for right-to-left links would be:

```
[ LRr [ML IGMiddle MR]* RLr ] (->) "{Rel-right-to-left"
                                   ... "Rel-right-to-left}"
                                        || IGHead _ IGDep RR
```

3. Since nothing in the format of the rules indicates the direction of the link, the direction would need to be indicated by the type of braces that are used to (temporarily) mark the segment claimed for the link. For instance, for a left-to-right rule to link a subject IG to a verb IG, we would use braces {`SBJ-left-to-right` and `SBJ-left-to-right`} and for a right-to-left rule (for the same kind of relation), we would use symbols {`SBJ-right-to-left` and `SBJ-right-to-left`}. The transducer that inserts the appropriate markers for links (`MarkChannels` in section 6.4) would then execute the appropriate action based on the type and the direction indication of the delimiting braces. For left-to-right braces it will insert the (lowercase) link start symbol to the right side of the left brace and the (uppercase) link end symbol to the left side of the right brace. For right-to-left braces, it will insert the link start symbol to the left side of the right brace and the link end symbol to the right side of the left brace.

4. The regular expressions checking the channel symbols around the dependent and head IGs would be different for the two types of rules. This is basically necessitated by the fact that since the IGs could now have links outgoing from both sides, checks have to be made on both sides:

   - LR_l in left-to-right rules would check that the dependent IG is a word-final IG and is not already linked and that no links are

crossing in or out. So it would function like LR, described in section 6.3.

- LL, just to the left of the IGDep pattern, would also make sure that the IG is not linked, via a right-to-left link, to an IG further to the left.
- RL_l would function just like RL, described in section 6.3.
- LR_r, which, for right-to-left rules, would be constraining the left channel symbols of the head IG, would need only to ensure that the top channel is available for a link and that no other links are crossing in and out.
- RL_r would ensure that the dependent IG is not linked to any IG to the left and that there are no links crossing, and that the top channel is available.
- RR, just to the right of IGDep in the right-to-left rule, would make sure that the dependent IG is not linked to any IG to the right and would additionally check that the top channel is available and that no links are crossing.

There is, however, a potential problem for a grammar with both left-to-right and right-to-left links. Robinson's axioms (see section 3) do not seem to disallow cyclic dependency links (unless the antisymmetry is interpreted to apply over the transitive closure of the "depends on" relationship), but configurations involving cycles are not assumed to correspond to legitimate dependency structures.

When both left-to-right and right-to-left links exists in a grammar, it is conceivable that two left-to-right rules may separately posit two left-to-right links, so that IG A links to IG B, IG B (or the word-final IG of the word to which B belongs) links to IG C, and later in a subsequent iteration, a right-to-left rule posits a link from IG C (or the word-final IG of the word to which C belongs) to IG A, where IG A precedes IG B, which precedes IG C in linear order. An implementation for a grammar would have to recognize such circular structures and eliminate them. It is possible to filter some of these cyclic configurations using a finite-state filter, but some will have to be checked later by a non-finite-state procedure.

If all but one of the links forming a cycle are established in the same channel (e.g., following the example above, the links from IG A to IG B and from IG B to IG C are established in the next-to-the-topmost channel), the cycle-forming link has to be established in the (current) topmost channel (that is, the right-to-left link from IG C to IG A has to be established there; otherwise the configuration will be filtered by the rule that says links have to established as the earliest possible channel). In order for a cycle to form in this case, IGs A, B, and C with have to be in sequential words, and the cycle-inducing link and the other links in the "other" direction will all be side by side. A set of simple regular expressions can recognize if a series of pairs of link start and link end symbols in one direction all appearing in the next-to-top channel (i.e., the second symbol to the left and right of (and), respectively, are surrounded by a link end–link start pair for the cycle-inducing link in the other direction) and kill any such configurations.

If, however, cycles are induced by links appearing in more than two different channels, then there is no elegant way of recognizing these in the finite-state framework, and such cases would have to be checked through other means.

6.6 Iterative Application of the Parser

Full parsing consists of iterative applications of the `Parser` transducer until a fixed point is reached. It should be noted that in general, different dependency parses of a sentence may use different numbers of channels, so all these parses have to be collected during each iteration.

Let `Sentence` be a transducer that represents the word sequence. The pseudocode for iterative applications of the parser is given as follows:

```
# Map sentence to a transducer representing a lattice of IGs
  M = [Sentence .o. MorphologicalAnalyzer];
# Initialize Parses
  Parses = { };
  i = 0;
  while (M.l != { } && i < MaxIterations) {
# Parse and filter the current M
     X = M .o. Parse .o. SyntacticFilter;
# Extract any configurations which correspond to parses
     Partial = X .o. OnlyOneUnlinked;
# and union with the Parses transducer
     Parses = [Parses | Partial];
# filter any stale configurations
     M = [X - Partial] .o. TopChannelNotEmpty;
     i = i+1;
}
```

Leaving the details of the transducer `SyntacticFilter`, a filter that eliminates configurations violating various linguistically motivated constraints, to a later section, this pseudocode works as follows: First, the sentence coded in `Sentence` is composed with the `MorphologicalAnalyzer`, which performs full morphological analysis of the tokens in the sentence along with some very conservative local morphological disambiguation. The resulting transducer encodes the sentence as a lattice representing all relevant morphological ambiguities. It is also possible to disambiguate the sentence prior to parsing with a tagger and present the parser with a fully disambiguated sentence.

During each iteration, `M` encodes as a transducer, the valid partial-dependency configurations. First `X` is computed by applying the `Parse` and `SyntacticFilter` transducers, in that order, to `M`. At this point, there may be some complete parses, that is, configurations that have all except one of their word-final IGs linked (e.g., a parse in which every IG is linked to the next IG would use only the first channel, and such a parse would be generated right after the first iteration.) The transducer `X` encoding the result of one iteration of parsing is filtered by `OnlyOneUnlinked`, defined as

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

This would be read as "It is not the case that there is more than one instance of word-final IGs whose right channel symbols do not contain any outgoing link marker."[18] This filter lets only those configurations that have all their required links established,

18 Note that this constraint is for a grammmar with left-to-right links.

that is, all word-final IGs, except one, are linked (only one word-final IG has all of its right channel symbols as 0s and 1s.) Any such parses in `Partial` are unioned with `Parses` (initially empty) and removed from `X` to give the `M` for the next iteration. Any configurations among the remaining ones (with no links in the most recently added channel, because of optionality in bracketing) are filtered, since these will violate the empty-channel constraint (see Figure 7(b)). This is achieved by means of composition, with the transducer `TopChannelNotEmpty` defined as follows:

```
TopChannelNotEmpty = ~[ ["<" LeftChannelSymbols* "0"
                          "(" AnyIG ("@") ")" "0"
                           RightChannelSymbols* ">" ] *] ;
```

This filter would be read as "It is not the case that all topmost channel symbols in a configuration are all 0s." Thus configurations in which *all* most recent channel symbols are `0` are filtered. If the lower language of `M` (denoted by `M.l`) becomes empty at this point (or we exceed the number of maximum number of iterations), the iteration exits, with `Parses` containing the relevant result configurations. `MaxIterations` is typically small. In the worst case, the number of iterations one would need would equal the number of word-final IGs, but in our experiments parsing has converged in five or six iterations, and we have used eight as the maximum.

6.7 Handling Coordinating Conjunctions

Headless constructions such as coordinating conjunctions have been one of the weaker points of dependency grammar approaches. Our implementation of coordinate conjunction constructs essentially follows the formulation of Järvinen and Tapanainen (1998). For a sequence of IGs like

$$D_1 \ldots C \ldots D_2 \ldots C \ldots\ldots D_k \ldots H$$

where D_i are the dependent IGs that are coordinated and C represents the conjunction IGs (for, (comma), *and*, and *or*), and H is the head IG, we effectively thread a "long link" (possibly spanning multiple channels) from D_1 to H. If the link between D_k and H is labeled L, then dependent D_i links to the following C with link L, and this C links to D_{i+1} with L. This is conceptually equivalent to the following: The "logical" link with label L from conjoined dependent X and Y to their head Z is implemented with three actual links of type L: X–*and*, *and*–Y, and Y–Z. If there are additional conjunctions and conjuncts, we continue to add (as required) one link of type L per word: Linking conjoined dependents (W and X and Y) to Z is implemented with links W–*and*, *and*–X, X–*and*, *and*–Y, and Y–Z.

One feature of Turkish simplifies this threading a bit: The left conjunct IG has to immediately precede the conjunction IG. The rules that do not involve conjunctions establish the link between D_k and H. For each such rule, we need two simple rules: The first rule links a dependent $D_i, (i < k)$, to the conjunction immediately following. Since the link type is almost always determined by the inflectional features of the dependent, this linking can be done (ambiguously in a very few cases in which dependent features do not uniquely determine the link type). The second rule links the conjunction to the right conjunct. Note that this applies only to conjunct IGs that have already been linked to from their left conjunct. Since the outermost link symbol on the left side of a conjunction IG identifies the relation (because the left conjunct is immediately to the left of this IG), the link emanating from the conjunction to the right can be made to land on an IG that agrees with the left conjunct in relevant features.

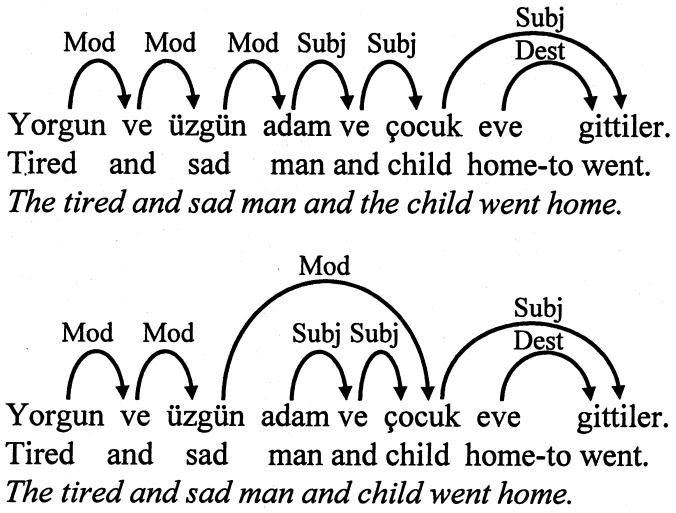


Figure 9
Link configurations for conjunction ambiguity.

When we have two groups of conjoined constructs

$$D_1\ C \ldots D_2\ C \ldots D_k \ldots H_1\ C \ldots H_2\ C \ldots H_l$$

the rightmost conjunct of the first group, D_k, will alternately attach to H_1, H_2, ..., H_l. In the first case, the complete first conjunction group links D_k to H_1, but not to the rest. In the second case, the complete first conjunction group links to the conjunction of H_1 and H_2, which are then conjoined with H_3 through H_l. In the last case, the complete first group links to the complete conjunction of H_1, H_2, ..., to H_l. In all cases, the links from H_1 all the way to H_l are independently threaded. A number of additional constraints also filter situations in which a conjoined head has both conjoined and locally attached dependents of the same type, by checking that the left channel symbols for these are not interleaved with other symbols. Figure 9 provides an example for this kind of conjunction ambiguity. In this implementation we have not attempted to handle circumscribing conjunctions such as the equivalents of *either ... or*.

6.8 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, more than one determiner or possessor may attach to a nominal, an object may attach to a verb that is then passivized in the next IG, or a nominative personal pronoun may be linked as a direct object (which is not possible in Turkish).

Some of the constraints preventing these configurations can be encoded in the bracketing rule patterns. For instance, a rule for linking a nominal IG to a verb IG as a subject may check, using a suitable regular expression, the left-hand channel symbols of the verb IG to make sure that it does not already contain an incoming subject link. There are also a number of situations in which the determination of a link depends on a pattern that is outside the sequence of the IGs from dependent to the head IG specified in a bracketing rule (but nevertheless in the same word in which the head IG is located). For instance, in Turkish, present participles are considered

modifiers derived from verbs. The verb part is the head of the sentential clause with a subject gap. Thus if a nominal IG attaches to a verb IG as a subject, but the verb IG is followed by another IG indicating that it is a present participle, then we should kill this configuration, since such verbs are not allowed to have subjects. It is also possible to incorporate almost all lexicalized argument structure–related constraints for dealing with intransitive and transitive verbs, provided the lexicon component (the morphological analyzer in our case) produces such lexically determined features.

We have chosen not to encode such constraints in the general format of the rules and to implement them instead as filters that eliminate configurations produced by the parsing. We have observed that this makes the linking rules more perspicuous and easier to maintain.

Each constraint is implemented as a finite-state filter that operates on the outputs of the `Parse` transducer by checking the symbols denoting the relations. For instance, we can define the following regular expression for filtering out configurations in which two determiners are attached to the same IG:

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

This regular expression constrains the form of the configurations generated by parsing. Note that this transducer lets through a sequence of zero or more IGs, none of which have more than one `D` symbol (indicating an incoming determiner link) among the left channel symbols. The crucial portion at the beginning of the regular expression 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 (that is, the intersection of the symbols between `<` and `(` with `LeftChannelSymbols` does not contain more than one `D`)."

We can provide the following finite-state filter as an example in which the violating configurations can be found by checking IGs following the head IG. For instance, the configurations in which subjects are linked to verbs which are then derived into present participles would be filtered by a finite-state filter like

```
NoSubjectForPresentPart = ~$[ "<" $["S"] & LeftChannelSymbols*
                                    "(" Verb ")"
                                       RightChannelSymbols* ">"
                                 "<" LeftChannelSymbols*
                                    "(" PresentParticipleIG ("@") ")"
                                       RightChannelSymbols* ">" ]
```

which says that the configuration does not contain, among the left-side channel symbols, a verb IG with a subject marker followed by a present participle IG.

The following are examples of the constraints that we have encoded as finite-state filters:

- At most one subject can link to a verb.
- At most one direct object can link to a verb.
- At most one dative (locative, ablative, instrumental) adjunct can link to a verb.
- Finite verbs derived from nouns and adjectives with zero suffixes do not have objects (as they are the equivalents of *be* verbs).

- Reflexive verbs do not get any overt objects.
- A postposition must always have an object to its immediate left (hence such a dependent nominal cannot link to some other head.)[19]

All syntactic constraints can be formulated similar to those given in the list. All such constraints `Cons1, Cons2 ...ConsN` can then be composed to give one transducer that enforces all of these:

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

In the current implementation we use a total of 28 such constraints.

6.9 Robust Parsing

It is possible that either because of grammar coverage, or because of ungrammatical input, a parse with only one unlinked word-final IG may not be found. In such cases, `Parses` in the pseudocode for parsing presented in section 6.6 would be empty. One might, 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 Karttunen's **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 an input string satisfy 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 that accepts parse configurations with no more than i unlinked word-final IGs. For instance, for $i = 2$, this would be defined as follows:

```
Unlinked_2 =  ~[[$[ "<" LeftChannelSymbols* "(" AnyIG  "@"  ")"
                      ["0" |   1]* ">"]]^ >  2  ];
```

which rejects configurations having more than two 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). We can augment the pseudocode given in section 6.6 as follows:

```
if (Parses == { }) {
   PartialParses = M .O. Unlinked_1 .O. Unlinked_2 .O.  Unlinked_3;
}
```

This will have the parser produce outputs with up to three 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. We have used this approach quite productively during the development of the dependency linking rules to discover coverage gaps in our grammar.

19 In fact, the morphological analyzer produces, for each postposition, a marker denoting the case of the preceding nominal as a subcategorization feature. This is used in a semilexicalized fashion while linking nominals to their head postpositions.

7. Experiments with Dependency Parsing of Turkish

Our implementation work has mainly consisted of developing and implementing the representation and finite-state techniques involved here, along with a nontrivial grammar component; we have not attempted to build a wide-coverage parser that is expected to work on an arbitrary test corpus. Although we have built the grammar component manually using a very small set of sentences, it is conceivable that future work on inducing (possibly statistical) dependency grammars will exploit dependency treebanks, which are slowly becoming available (Hajič 1998; Oflazer et al. 2003).

The grammar has two major components. The morphological analyzer is a full-coverage analyzer built using XRCE finite-state 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 the pseudocode given in section 6.6), a transducer for the lattice representing all IGs for all morphological ambiguities (remaining after a light morphological disambiguation) is generated. The dependency relations are described by a set of about 60 rules much like the ones exemplified earlier. These rules were developed using a small set of 30 sentences. The rules were almost all nonlexical, establishing links of the types listed earlier. There is an additional set of 28 finite-state constraints that impose various syntactic and structural constraints. The resulting `Parser` transducer has 13,290 states and 186,270 transitions, and the `SyntacticFilter` transducer has 3,800 states and 134,491 transitions. The combined transducer for morphological analysis and (very limited) disambiguation has 100,103 states and 243,533 arcs.

The dependency grammar and the finite-state dependency parser were tested on a set of 200 Turkish sentences, including the 30 that were used for developing and testing the grammar. These sentences had 4 to 43 words, with an average of about 18 words. Table 1 presents our results for parsing this set of 200 sentences. This table presents the minimum, the maximum, and the average of the number of words and IGs per sentence, the number of parser iterations and the number of parses generated. (The number of iterations includes the last iteration where no new links are added.) There were 22 sentences among the 200 that had quite a number of verbal adjuncts that function as modifiers. These freely attach to any verb IG, creating an analog of the PP attachment problem and giving rise to a very large number of parses. The last row in the table gives the minimum, maximum and the average number of parses when such sentences were not considered.

To impose a ranking on the parses generated based on just structural properties of the dependency tree, we employed Lin's (1996) notion of structural complexity. We measured the total link length (TLL) in a dependency parse counting the IGs the links pass over in the linear representation and ordered the dependency parses based on the TLL of the dependency tree. We classified the sentences into six groups:

1. *Sentences that had a single minimum TLL parse which was correct.* There were a total of 39 sentences (19.5%) in this group.

2. *Sentences that had more than one parse with the same minimum TLL and the correct parse was among these parses.* There were 58 sentences (29.0%) in this group. Thus for a total of 97 (48.5%) sentences, the correct parse was found among the parses with the minimum TLL. In these cases the average number of parses with the minimum TLL was about 6

(minimum 1 parse and maximum 38 parses with the same minimum TLL).

3. *Sentences for which the correct parse was not among the minimum TLL parses but was among the next-largest TLL group.* There were 29 (14.5%) sentences in this group.

4. *Sentences for which the correct parse was not among the smallest and the next-smallest TLL groups, but among the next three smallest TLL groups.* There were a total of 26 (13%) sentences in this group.

5. *Sentences for which the parser generated parses, but the correct parse was not among the first five groups.* There were 26 (13%) such sentences. For these, we did not check any further and assumed there were no correct parses. The parses that were generated usually used other (morphological) ambiguities of the lexical item to arrive at a parse.

6. *Sentences for which no parses could be found, usually as a result of the lack of coverage of the dependency grammar and the morphological analyzer.* There were 22 (11%) sentences in this group.

It seems that for quite a number of sentences (groups 1–3 in the list), a relatively small number of parses have to be processed further with any additional lexical and/or statistical constraints to extract the correct parse. Although to obtain the statistics in items 1–6, we had to extract the full set of parse strings from the transducer that encoded the parses compactly, one does not have to do this. The parses with the shortest link length can be found by treating the resulting parse lattice transducer as a directed acyclic graph and finding the path with the minimum number of `1` symbols on it from the start state node to the final state node using one of the standard shortest-path algorithms (e.g., Dijsktra's algorithm [Cormen, Leiserson, and Rivest 1990]).[20] This is because paths from the start state to the final state are string encodings of the dependency trees. The `1` symbols in the representation add up to the total link length of the encoded dependency tree. Since the representation of the tree is quite convoluted, the `1`s in a block of `1`s in the string representation all belong to different links stacked on top of each other. Thus we "count" the length of the links in an "interleaved" fashion. On the other hand, Dijkstra's algorithm may not be very useful, since one may need to extract the k shortest paths to select from, perhaps, later, with more-informed criteria than link length, such as lexical and statistical information. For this we may use an algorithm which finds the k shortest paths between a source and a sink node in a directed graph (e.g., Eppstein 1998).

The complete parser, including about 60 linking rules and the 28 syntactic constraints, is defined using about 240 regular expressions coded using XRCE regular expression language. These regular expressions compile in about one minute on Pentium III 700 MHz (running Linux) into the `Parser` and `SyntacticFilter` transducers. The parser iterations are handled by a script interpreted by the XRCE finite state tool, *xfst*.

Parsing takes about a second per sentence, including lookup in the morphological analyzer, which is performed with a composition. With manually completely morphologically disambiguated input, parsing is essentially instantaneous.[21]

20 I thank an anonymous reviewer for suggesting this.
21 We performed a simple experiment with 14 sentences that were manually morphologically

Table 1
Statistics from parsing 200 Turkish sentences.

	Minimum	Maximum	Average
Words/sentence	4	43	18.2
IGs/sentence	4	59	22.5
Parser iterations	3	8	5.0
Parses/sentence	1	12,400	408.7
Parses/sentence excluding the 22 sentences with $>$ 1,000 parses	1	285	55.4

Input Sentence: Dünya Bankası Türkiye Direktörü hükümetin izlediği ekonomik programın sonucunda önemli adımların atıldığını söyledi.
English: The World Bank Turkey director said that as a result of the economic program followed by the government, important steps were taken.
Parser output after three 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>small
```

Figure 10
Sample input and output of the parser. The only difference in the two parses is in the locative adjunct attachment (to verbs *at* and *söyle*). The IGs that differ in the two parses are `<S11(at+Verb)110>` versus `<SL1(at+Verb)100>`, and `<OLS(sOyle+Verb+Pos+Past+A3sg@)000>` versus `<OOS(sOyle+Verb+Pos+Past+A3sg@)000>`.

Figure 10 presents the input and the output of the parser for a sample Turkish sentence: *Dünya Bankası Türkiye Direktörü hükümetin izlediği ekonomik programın sonucunda önemli adımların atıldığını söyledi*. (The World Bank Turkey director said that as a result of the economic program followed by the government, important steps were taken.) Figure 11 shows the output of the parser processed with a Perl script to provide a more human-readable presentation:

disambiguated. For this set of sentences, there were about 7 parses per sentence. The average number of parses for these sentences when all their morphological ambiguities were considered was 15. When the two sentences with the highest number of parses were removed from this set, the corresponding numbers were 3 parses per sentence and 11 parses per sentence.

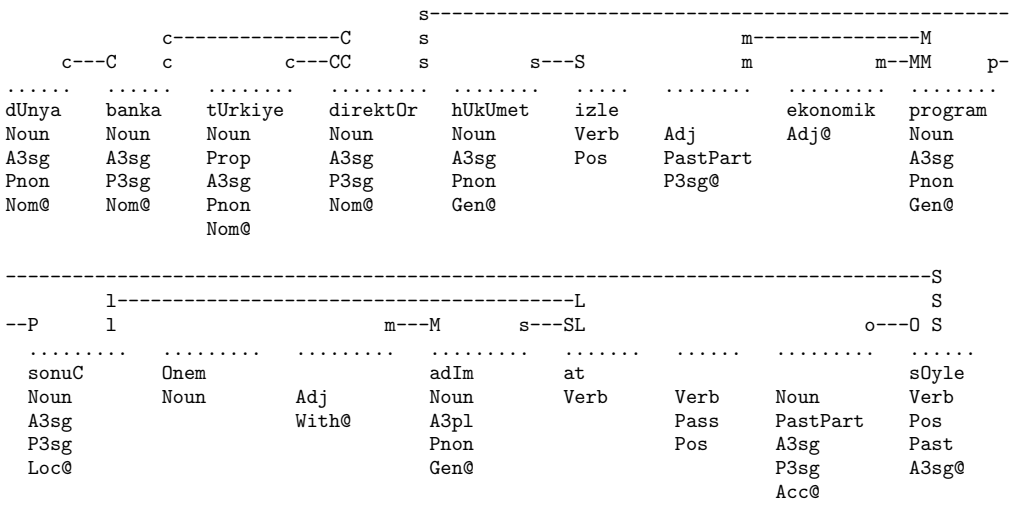


Figure 11
Dependency tree for the second parse.

8. Discussion and Conclusions

We have presented the architecture and implementation of a dependency parser using an extended finite state. Although the emphasis has been on the description of the approach, we have developed a dependency grammar for Turkish and have used it to experiment with a small sample of 200 Turkish sentences. We have also employed a scheme for ranking dependency parses using the total link length of the dependency trees, as originally suggested by Lin (1996), with quite promising results. It is possible to use algorithms for extracting k shortest paths to extract parses from the transducer, which compactly encodes all dependency parses, and further to rank a much smaller set of parses using lexical and statistical information whenever available.

Another interesting point that we have noted, especially during the development of the grammar, is that the grammar rules do not have to pay any real attention to the sequence of the IGs that do not have anything to do with the current rule (with a very few exceptions in some special cases in which the rules have to check that links do not cross a "barrier"). This means that that the grammar of the IG sequence is really localized to the morphological analyzer and that for the most part the dependency grammar does not have to "know" about the sequencing of the IGs within a word.

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 from optionality in bracketing, mainly by enforcing obligatory bracketing for sequential dependent-head pairs (e.g., the complement of a postposition is immediately before it, or for conjunctions, the left conjunct is always the previous IG). Such heuristics force such dependencies to appear in the first channel and hence prune many potentially useless configurations popping up in later iterations. Although we have not performed any significant experiments with the robust parsing technique that we describe in the article, it has been very instrumental during the process of debugging the grammar. During debugging, when the actual parser did not deliver any results after a certain number of iterations, we generated partial parses with up to four unlinked word-final IGs to see where we were having problems with the coverage and added new linking rules.

Acknowledgments

This work was partially supported by grant EEEAG-199E027 from TÜBİTAK (The Turkish Council on Scientific and Technical Research). A portion of this work was done while the author was visiting the Computing Research Laboratory at New Mexico State University. The author thanks Lauri Karttunen of Xerox PARC for making available XRCE finite-state tools. Mercan Karahan, currently of Purdue University, helped substantially with the implementation of the parser and with the experimentation. Comments by anonymous reviewers helped substantially to improve the article.

References

Abney, Steven. 1996. Partial parsing via finite-state cascades. *Journal of Natural Language Engineering*, 2(4):337–344.

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

Black, Alan. 1989. Finite state machines from feature grammars. In *Proceedings of International Workshop on Parsing Technologies*, pages 277–285.

Chanod, Jean-Pierre and Pasi Tapanainen. 1996. A robust finite-state grammar for French. In John Carroll and Ted Briscoe, editors, *Proceedings of the ESSLLI'96 Workshop on Robust Parsing*, pages 16–25, August.

Chelba, Ciprian, David Engle, Frederick Jelinek, Victor Jimenez, Sanjeev Khudanpur, Lidia Mangu, Harry Printz, Eric Ristad, Ronald Rosenfeld, Andreas Stolcke, and Dekai Wu. 1997. Structure and estimation of a dependency language model. In *Processings of Eurospeech'97*.

Collins, Michael. 1996. A new statistical parser based on bigram lexical dependencies. In *Proceedings of the 34th Annual Meeting of the Association for Computational Linguistics*, pages 184–191.

Collins, Michael, Jan Hajič, Lance Ramshaw, and Christoph Tillman. 1999. A statistical parser for Czech. In *Proceedings of the 37th Annual Meeting of the Association for Computational Linguistics*, pages 505–512, June.

Cormen, Thomas H., Charles E. Leiserson, and Ronald L. Rivest. 1990. *Introduction to Algorithms*. MIT Press, Cambridge, MA.

Eisner, Jason. 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.

Elworthy, David. 2000. A finite state parser with dependency structure output. In *Proceedings of International Workshop on Parsing Technologies*.

Eppstein, David. 1998. Finding *k*-shortest paths. *Siam Journal on Computing*, 28(2):652–673.

Giguet, Emmanuel and Jacques Vergne. 1997. From part-of-speech tagging to memory-based deep syntactic analysis. In *Proceedings of the International Workshop on Parsing Technologies*, pages 77–88.

Grefenstette, Gregory. 1996. Light parsing as finite-state filtering. In *ECAI '96 Workshop on Extended Finite State Models of Language*, August.

Grimley-Evans, Edmund. 1997. Approximating context-free grammars with a finite-state calculus. In *Proceedings of ACL-EACL'97*, pages 452–459.

Hajič, Jan. 1998. Building a syntactically annotated corpus: The Prague Dependency Treebank. In Eva Hajicova, editor, *Issues in Valency and Meaning: Studies in Honour of Jarmila Panenova*. Karolinum–Charles University Press, Prague, pages 106–132.

Hankamer, Jorge. 1989. Morphological parsing and the lexicon. In W. Marslen-Wilson, editor, *Lexical Representation and Process*. MIT Press, Cambridge, MA, pages 392–408.

Hobbs, Jerry R., Douglas Appelt, John Bear, David Israel, Megumi Kameyama, Mark Stickel, and Mabry Tyson. 1997. FASTUS: A cascaded finite state transducer for extracting information from natural language text. In Emmanuel Roche and Yves Schabes, editors, *Finite State Language Processing*. MIT Press, Cambridge, MA, pages 386–406.

Hopcroft, John E. and Jeffrey D. Ullman. 1979. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, MA.

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

Johnson, Mark. 1998. Finite state approximation of constraint-based grammars using left-corner grammar transforms. In *Proceedings of COLING-ACL'98*, pages 619–623, August.

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

Karttunen, Lauri. 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.

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

Kokkinakis, Dimitrios and Sofie Johansson Kokkinakis. 1999. A cascaded finite state parser for syntactic analysis of Swedish. In *Proceedings of EACL'99*.

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

Koskenniemi, Kimmo, 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.

Lafferty, John, 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*.

Lai, Bong Yeung Tom 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*.

Lin, Dekang. 1995. A dependency-based method for evaluation of broad-coverage parsers. In *Proceedings of IJCAI'95*.

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

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

Mohri, Mehryar, Fernando C. N. Pereira, and Michael Riley. 1998. A rational design for a weighted finite-state transducer library. In Derick Wood and Sheng Yu, editors, *Proceedings of the Second International Workshop on Implementing Automata (WIA '97)*, volume 1436 of Lecture Notes in Computer Science. Springer-Verlag, Berlin, pages 144–158.

Nederhof, Mark-Jan. 1998. Context-free parsing through regular approximation. In Lauri Karttunen and Kemal Oflazer, editors, *Proceedings of International Workshop on Finite State Methods in Natural Language Processing*, pages 13–24, Ankara, Turkey.

Nederhof, Mark-Jan. 2000. Practical experiments with regular approximation of context-free languages. *Computational Linguistics*, 26(1):17–44.

Oflazer, Kemal. 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*, 9(2), 1994).

Oflazer, Kemal, Bilge Say, Dilek Zeynep Hakkani-Tür, and Gökhan Tür. 2003. Building a Turkish treebank. In Anne Abeillé, editor, *Treebanks*. Kluwer Academic Publishers, Dordrecht, the Netherlands.

Pereira, Fernando C. N., and Rebecca N. Wright. 1997. Finite state approximation of phrase structure grammars. In Emmanuel Roche and Yves Schabes, editors, *Finite State Language Processing*. MIT Press, Cambridge, MA.

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

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

Roche, Emmanuel and Yves Schabes, editors. 1997. *Finite State Language Processing*. MIT Press, Cambridge, MA.

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

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

van Noord, Gertjan. 1997. FSA utilities: A toolbox to manipulate finite-state automata. In Derick Wood, Darrell Raymond, and Sheng Yu, editors, *Automata Implementation*, volume 1260 of Lecture Notes in Computer Science. Springer-Verlag, Berlin.

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