Bracketing Input for Accurate Parsing *

Yongkyoon No[a]

Chungnam National University
220 Koong-dong, Yuseong-gu, Taejeon, Korea
yno@linguist.cnu.ac.kr

Abstract. Syntax parsers can benefit from speakers' intuition about constituent structures indicated in the input string in the form of parentheses. Focusing on languages like Korean, whose orthographic convention requires more than one word to be written without spaces, we describe an algorithm for passing the bracketing information across the tagger to the probabilistic CFG parser, together with one for heightening (or penalizing, as the case may be) probabilities of putative constituents as they are suggested by the parser. It is shown that two or three constituents marked in the input suffice to guide the parser to the correct parse as the most likely one, even with sentences that are considered long.

Keywords: manually parsed corpus, Probabilistic Context Free Grammar, Korean syntax, bottom-up chart parser, pre-annotated input, Paak, KWGInterpreter

1. Introduction

Syntactic parsers often produce parses many of whose parts are correct but which do contain partially inaccurate structures. To attain a high level of precision and recall, serious research has to be conducted and just about every kind of research in this field may well benefit from a moderate amount of manually parsed, i.e, correctly parsed, sentences.

When it comes to the Korean language, a widely available, yet dependable corpus of parsed sentences is something that has yet to be constructed. In this paper, we describe a move that will help build a parsed corpus of about one thousand sentences. Our idea here is to use an initial stage probabilitic CFG parser with a partially bracketed input. This work is inspired by Pereira and Schabes (1992), who use bracketed sentences to induce grammar rules automatically.

We will describe our initial stage parser, *Paak*, in Section 2. How the input string with some bracketings are processed in this parser is described in the next section. Section 4 gives algorithms for keeping track of the bracketing information and for affecting the probabilities of constituents, which conform to, are consistent with, or contradict the information annotated in the input. The performance gains at each degree of bracketing information are estimated in Section 5, where it is shown that no more than two pairs of bracketings suffice to yield the correct parse of a sentence of length twenty.

2. The probabilistic CFG parser *Paak*

Paak is a probabilistic parser of the Korean language with a set of context-free grammar rules. It relies on a tagger of Korean for its analysis of input string into the smallest units of syntactic

* I am grateful to one of the anonymous referees of the Program Committee of PACLIC 21, who gave me useful comments.

processing.[1] The tagger uses fifty three tags, many of which correspond to parts-of-speech. *Paak* has 45 preterminal symbols and 20 nonterminals. Currently it has just over three hundred PCFG rules.

The parser is a probabilistic parser implemented in Java. The implementation basically follows Charniak(1993: 1.4). It is a bottom up parser. Its grammar rules have been extracted from a very small hand-parsed corpus of twenty sentences.

3. Adding constituent structures to the input string

The parser takes an input file and gives an output file. Its input file normally consists of sentences of the language. What we suggest is add information about the input sentence's constituent structures so as to guide the parser to its correct parse.

The input string itself may have bracketings added to them. Adding bracketings to a string of words would be the easiest way of annotating their syntactic structure.

(1) a. 일상생활에 지장이 있는 정도는 아니라지만, 오른쪽 목과 어깨의 통증으로 고생이 심하다.
b. (일상생활에 지장이 있는 정도는 아니라)지만, (오른쪽 목과 어깨의 통증)으로 고생이 심하다.
c. (일상생활에 지장이 있는 정도는 아니라)지만, (오른쪽 (목과 어깨)의) 통증으로 고생이 심하다.
d. (((((((일상생활)에) (지장이) 있는) 정도)는 아니라)지만), ((((오른쪽 ((목과) 어깨))의) 통증)으로) ((고생이) 심하다).)

The first string above, (1a), has no information added. It is the kind of input string ordinary parsers would expect. (1b) has two constituents marked with pairs of parentheses. The next string has three constituents marked and the last, (1d) has all its nonlexical constituents marked. None of these strings mark their constituents with labels, however.

We are to allow our parser to process all of the input types given in (1). A special provision has to be made in order to distinguish the ordinary characters `(' and `)' from our metacharacters for marking the boundaries of constituents.

The parser ought to identify the metacharacters and their positions and then has to remove them before it passes them to the tagger. The parser will make use of the partial constituency information in calculating the input's constituents. If the constituent being built by the parser at a given point is one of the spans indicated by the bracketings, its probability is to be heightened. If the constituent being built is in conflict with a bracketing marked in the input, its probability is to be lowered. As the probability of a parse is the product of the probabilities of all its parts, making a constituent's probability higher or lower directly affects the probability the parse of the whole sentence ends up getting.

4. Algorithms

Getting the parser to make use of preannotated constituency involves two sorts of algorithms. The first is to identify the positions of the opening and closing brackets in such a way that they correspond to boundaries of the words to be fed to the parser. As the words to be fed to the

[1] The lemmatizing POS tagger of the language is being serviced at `http://linguist.cnu.ac.kr:8080/servlets/KWGInterpreter`. Its properties are described in No(2007).

parser are only available after the tagger performs analysis on the input string, and as the bracketings need to be removed before the input string is submitted to the tagger, the task of assigning correct positions to the brackets is not trivial.

The second algorithm concerns affecting the probabilities of constituents built by the initial stage parser with the constituency information gathered by our first algorithm. We describe these in turn.

4.1 Mapping positions across the tagger

The Korean language has many clitics. Its orthographic convention allows much freedom in tokenizing its words. Sequences of words are required to be written without intervening spaces and some sequences are allowed to be written with or without intervening spaces. Thus, sentence (1a) would have eleven groups in ordinary writings, as their boundaries are indicated with integers in (2).

However, as many of these groups have more than one word in them, the result of tagging the sentence would be twenty three words. This situation is illustrated with integers indicating word boundaries in (3).

(2) 0 일상생활에 1 지장이 2 있는 3 정도는 4 아니라지만, 5 오른쪽 6 목과 7 어깨의 8 통증으로 9 고생이 10 심하다. 11

(3) [0] 일상 [1] 생활 [2] 에 [3] 지장 [4] 이 [5] 있는 [6] 정도 [7] 는 [8] 아니라 [9] 지만 [10] , [11] 오른 [12] 쪽 [13] 목 [14] 과 [15] 어깨 [16] 의 [17] 통증 [18] 으로 [19] 고생 [20] 이 [21] 심하다 [22] . [23]

The task for the parentheses identifier, let us call it **g2w**, is to map 0 to [0], 1 to [3], 2 to [5], and so on, given the tokenization in (2). This is straightforward when parentheses appear only at group boundaries: the tagger analyzes each group of words into words, the number of which is easy to count. The parenthesis at position *p* of the input string marks position [$\sum_{i=0}^{p-1}$ number-of-words in g_i] of the output of the tagger (or, of the input to the parsing component).

However, many syntactic boundaries may well fall within a group. The group *ilsangsaynghwaley*(*일상생활에*) has a constituent boundary before the postposition *ey*(*에*), *cengtonun*(*정도는*) has one between the noun *cengto*(*정도*) and the delimiter *nun*(*는*), the group *anilaciman*(*아니라지만*) contains two verbs, the second verb taking a clause as its complement; only the noun in the noun-postposition group *ekkayuy*(*어깨의*) is a part of the conjunction *mokkwa ekkay*(*목과 어깨*), and hence, the group has a boundary before the postposition; similarly for *thoncungulo*(*통증으로*).

Group-internal parentheses present a difficulty in specifying **g2w**: the portion flanked by two boundaries (i.e., parentheses or a space) may turn out to be any number of word tokens. We avoid this difficulty by imposing the following constraint on the user:

Constraint
Group-internal parentheses are assumed to be dense. An internal parenthesis is interpreted as separating the *n*-th word from (*n* + 1)-th, only when there are *n* – 1 preceding group-internal parentheses.

Put in other words, the first parenthesis after the beginning of a group is taken to be marking the boundary between the first word of the group and the following words. The second parenthesis is taken to be marking the boundary between the second word of the group and the following words, and so on.

While group-internal boundaries may mark the end of a phrase, it does not seem to be the case that there are group-internal boundaries that mark the beginning of a phrase. The orthographic conventions of the language are such that all beginnings of phrases fall at a group boundary.[2]

This finding makes it easier to specify the boundaries-mapping function **g2w**.

1.Initialize a string, **trimmedGroup**, a stack, **aStack** and a list, **aList**.
2. Split the input string (possibly with brackets) into an array of strings, using the space character as the pivot.
3. For each string in the array, count the number of closing parentheses in it and split it into a second-order array, this time using the regular expression [)]*, as the pivot.
4. For each position *i* in the first-order array, for each position *j* in the second-order array do:
   (i) for each opening parenthesis in array[*i*][*j*] create a span (*i*, *j*, -1, -1) and push it onto **aStack**.
   (ii) for each closing parenthesis in array[*i*][*j*] pop a span from the stack. Set the span's third and fourth arguments to be *i* and *j*, respectively. Store the resulting span in **aList**.
   (iii) strip all opening parentheses off of array[*i*][*j*] and concatenate it with **trimmedGroup**.
5. Add **trimmedGroup** to the array of strings to be submitted to the tagger.

When the loop above has iterated through the input, **aList** ends up with as many spans as there are pairs of brackets in the input. The annotation-free input string, built by concatenations of **trimmedGroup**, is passed to the tagger, which outputs word tokens of the following form:

(4) (MassNoun 일상 daily_life) (ProcessNoun 생활 NA) (PO 에 GR) (MassNoun 지장 interference) (PN 이 GR) (vrel 있 exist) (VAdjunctNoun 정도 extent) (Delimiter 는 GR) (viq 아니 be_not) (vadv EMPTYSTEM say) (COMMA , comma) …

The tagger organizes word tokens into groups, as is dictated by the original input's groupings of words. Thus, how many words there are in each group can be kept in an array. Call it **convert**. The integer in **convert**[*k*] indicates that the (*k* + 1)-th group of the original input has this many words in it.

With the help of **convert**, the first and third terms of each span in **aList** are now updated. Each span, <start, startoffset, end, endoffset> is replaced with a new span, <convert[start], startoffset, convert[end], endoffset>.

4.2 Making the probabilities of constituents sensitive to bracketings

A bottom-up probabilistic parser calculates the probability of each constituent based on its subconstituents' probabilities. It simply multiplies over all its subconstituents' probabilities. We modify this part of the parser's behavior in such a way that it multiplies the resulting probability with a huge number in case the constituent to be formed matches one of the spans in the list of the designated spans.[3] A constituent matches a span iff its start position is the same as the latter's (start + startoffset) and its end position is the same as the latter's (end + endoffset).

A spurious constituent can be penalized by a further move of dividing, rather than multiplying, its probability with a huge number. Spurious constituents are a proper subset of constituents that

[2] The only exception seems to be the ones involving opening quotation marks.

[3] The probability of the putative constituent would be between 0 and 1. If this constituent is among the constituents marked in the input, its probability is to be multiplied by 1000000. It might suffice to give such a constituent the probability of 1.0.

fail to match a span in the list. As the annotations in the input string are partial, there could be constituents that do not match any of the spans but are indeed licit.
The constituents to be penalized are then those which conflict with a span in the list. A constituent c conflicts with a span s if either (i) c starts before s and ends in the middle of s or (ii) c starts in the middle of s and ends after s.[4]
These are cases of crossing brackets used as a measure of parser correctness. See Carroll, Briscoe and Sanfilippo (1998) and Jurafsky and Martin (2000: 464).

5. Gains of annotated input

We tested the modified parser with six sentences of various lengths and differing levels of annotation. The sentences are:

[A] kimkunthaynun cikumto komunhwuyucungul alhnunta.
김근태는 지금도 고문후유증을 앓는다.
[B] kunun kokaylul cayensulepkey tollici moshanta.
그는 고개를 자연스럽게 돌리지 못한다.
[C] hanchamul tallita poni eyncini simhakey ttellinta.
한참을 달리다 보니 엔진이 심하게 떨린다.
[D] kulena hakkyochukun uyhokul cosahaci anhko sinssilul cokyoswulo thukchayhayssta.
그러나 학교측은 의혹을 조사하지 않고 신씨를 조교수로 특채했다.
[E] hakkyochukun ohilye 3 kaywel twi yellin 228 cha isahoyeyse cangyunsunimul isacikeyse
학교측은 오히려 3 개월 뒤 열린 228 차 이사회에서 장윤스님을 이사직에서
hayimhayssta.
해임했다.
[F] ilsansaynghwaley cicangi issnun cengtonun anilaciman, olunccok mokkwa ekkayuy
일상생활에 지장이 있는 정도는 아니라지만, 오른쪽 목과 어깨의
thongcungulo kosayngi simhata.
통증으로 고생이 심하다.

Table 1, in the following page, gives the result.

It can be seen from the table that two pairs of bracketings are enough to guide the parser to the unique correct parse, in most of the cases. The only sentence the parser fails to give the correct parse as its most likely parse is Sentence F in Table 1, which is our example (1). Marking two of its constituents in its input, as in (\ref{neckshoulder}b), produces a wrong parse. See (6) below. Marking three, as in (1c), is enough. The most likely parse turns out to be the correct one, important part of which is shown in (7). In both parses, the sentential adjunct *ilsangsaynghwaley cicangi issnun cengtonun anilaciman*(*일상생활에 지장이 있는 정도는 아니라지만*) is assigned the same structure, which is judged correct. We present only the main clause portions, namely the subtree rooted by the lower **rootclause** node in (5).

Table 1. Number of total parses and rank of the correct parse

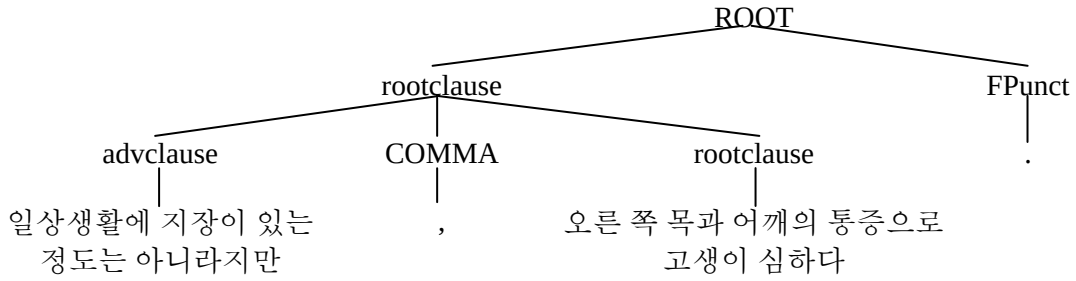
[4] As a span has offsets stored in it in addition to its start and end positions, c conflicts with s in either of the following cases:

(i) c.start < s.start + s.startoffset ∧ s.start + s.startoffset < c.end∧ c.end < s.end + s.endoffset
(ii) c.start > s.start + s.startoffset ∧ s.end + s.endoffset > c.start ∧ c.end > s.end + s.endoffset

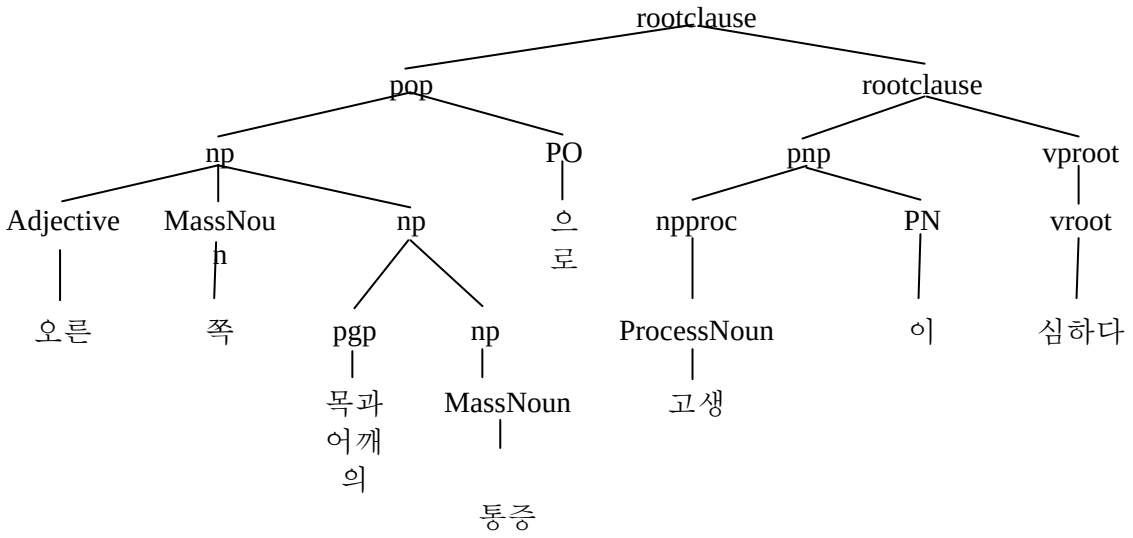
(b=*k* stands for the number of bracket pairs in the input; * indicates either no parse produced or no attempt to parse made)

sentence	# of groups	# of words	Total parses			Rank of correct parse		
			b=0	b=1	b=2	b=0	b=1	b=2
A	4	8	2	*	*	1	*	*
B	5	10	34	220	172	8	10	1
C	6	9	22	13	*	7	1	*
D	8	17	*	20	*	*	1	*
E	11	21	*	38	*	*	1	*
F	11	23	*	*	1020	*	*	3

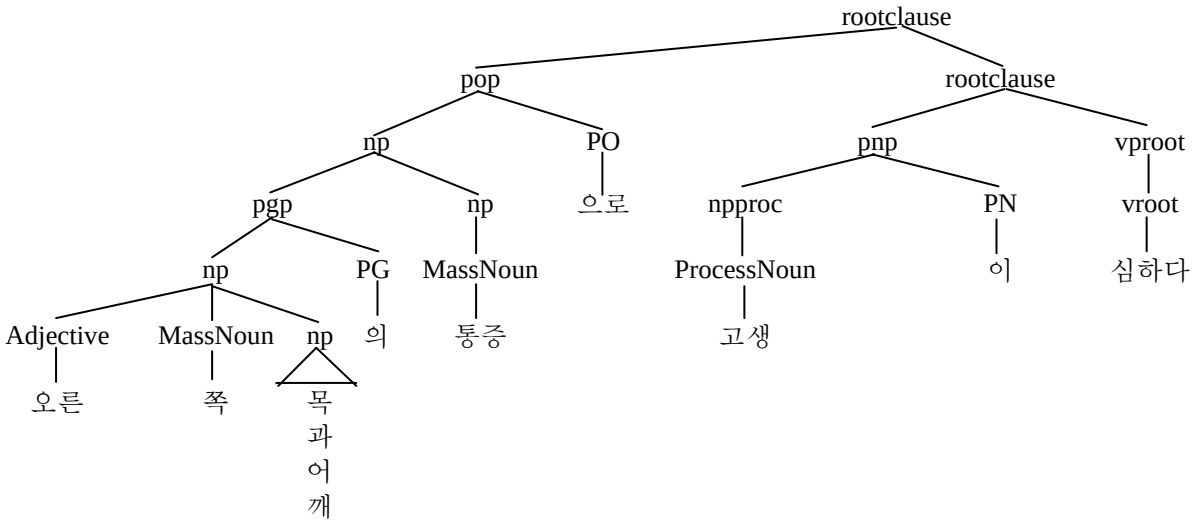
(5)



(6)



(7)



6. Conclusion

We have shown that accurate parses of the kind one would like to have in a manually parsed corpus can be obtained with bracketings added to the input string. The somewhat involved work of keeping track of bracketings in a language like Korean whose orthographic convention requires a sequence of words written in a single group, has been carried out. The number of constituents to be annotated in this fashion has proven small, which is good news to developers of parsed corpora. A parser, such as *Paak*, with this functionality certainly facilitates development of moderate-sized corpora, which in turn will be used to get better probabilities of its CFG rules.

References

Carroll, John, Ted Briscoe, and Antonio Sanfilippo. 1998. Parser Evaluation: A Survey and a New proposal. In *Proceedings, First International Conference on Language Resources and Evaluation*, pp. 447--54. European Language Resources Association.

Charniak, Eugene. 1993. *Statistical Language Learning*. The MIT Press.

Jurafsky, Daniel and James~H. Martin. 2000. *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*. Prentice-Hall, Inc.

No, Yongkyoon. 2007. KWGInterpreter: a lemmatizing POS tagger for the Korean language. *Proceedings of the 2007 Joint Conference of LAK, MLSK, and KSLI*, pp. 86--94. Linguistic Association of Korea.

Pereira, Fernando and Yves Schabes. 1992. Inside-outside Reestimation from Partially Bracketed corpora. In *27th Annual Meeting of the Association for Computational Linguistics*, . 128--135. ACL.