1

Parsing Partially Bracketed Input

MARTIJN WIELING, MARK-JAN NEDERHOF AND
GERTJAN VAN NOORD

Humanities Computing
University of Groningen

Abstract

A method is proposed to convert a Context Free Grammar to a Bracket Context Free Grammar (BCFG). A BCFG is able to parse input strings which are, in part or whole, annotated with structural information (brackets). Parsing partially bracketed strings arises naturally in several cases. One interesting application is semi-automatic treebank construction. Another application is parsing of input strings which are first annotated by a NP-chunker.

Three ways of annotating an input string with structure information are introduced: identifying a complete constituent by using a pair of round brackets, identifying the start or the end of a constituent by using square brackets and identifying the type of a constituent by subscripting the brackets with the type. If an input string is annotated with structural information and is parsed with the BCFG, the number of generated parse trees can be reduced. Only parse trees are generated which comply with the indicated structure.

An important non-trivial property of the proposed transformation is that it does not generate spurious ambiguous parse trees.

Proceedings of the Sixteenth Computational Linguistics in the Netherlands.
Edited by: Khalil Sima'an, Maarten de Rijke, Remko Scha and Rob van Son.

1.1 Introduction

Natural language is highly ambiguous. If natural language sentences are parsed according to a given Context Free Grammar (CFG), the number of parse trees can be enormous. If some knowledge about the type and coherence of words in a sentence is available beforehand, the number of parse trees can be reduced drastically, and the parser will be faster. In this paper we present a method to parse partially bracketed input.

The method presented in this paper is useful for a number of different applications. One interesting application is semi-automatic treebank construction. Another application is parsing of input strings which are first annotated by a syntactic chunker.

In recent years much effort is devoted to the construction of treebanks: sets of naturally occurring sentences that are associated with their correct parse. Typically, such treebanks are constructed in a semi-automatic way in which the sentence is parsed by an automatic parser, and a linguist then selects, and sometimes manually adapts, the appropriate parse from the set of parses found by the parser. If a sentence is very ambiguous this process is rather cumbersome and time consuming. In our experience in the context of the construction of the Alpino and D-Coi treebanks (van der Beek, Bouma, Malouf and van Noord 2002, van Noord, Schuurman and Vandeghinste 2006), the ability to add *some* brackets (possibly with the corresponding category) is a very intuitive and effective way to reduce annotation efforts.

Below, we also introduce the possibility to annotate a sentence with an opening bracket without a corresponding closing bracket, and vice versa. This possibility is motivated by the second application: parsing input that is pre-processed by a chunker. A chunker is an efficient program which finds occurrences of some syntactic categories (typically noun phrases). If a reliable and efficient chunker is available, syntactic parsing can be faster by using that chunker in a preprocessing stage. One common implementation strategy which goes back to Ramshaw and Marcus (1995) is to use techniques originally developed for POS-tagging, and to encode the start and end of chunks in the POS-tag inventory. Such chunkers are able to detect where a chunks starts, or where a chunk ends, but the fact that the beginning and the end of a chunk are supposed to co-occur is not inherent to the technique, but is usually added as an ad-hoc filter on the output. The ability of our method to allow independent opening and closing brackets in the input implies that this ad-hoc filter is no longer needed. It remains to be investigated if this improvement has empirical benefits as well.

In the past, researchers have experimented with techniques where pairs of parentheses are used to group constituents of an input string, such that fewer parse trees are generated. In Pereira and Schabes (1992) as well as Stolcke (1995) a method is given to adapt an existing parse algorithm (inside-outside and Earley) in such a way that it works faster with input strings which are annotated with pairs of parentheses. In McNaughton (1967) and Knuth (1967) features of a parenthesis grammar are discussed where brackets are added at the start and end of every production rule, $A \rightarrow (\ a \ )$. In their bracketed context free grammar, Ginsburg and Harrison (1967) add additional information by subscripting

brackets with unique indexes, $A \rightarrow [_1\ a\ ]_1$ and $A \rightarrow [_2\ b\ ]_2$.

In our research, we have focused on finding an automatic procedure to convert a given CFG to a Bracket Context Free Grammar (BCFG). A BCFG can parse the same input strings as the CFG, but in addition these input strings may be annotated in part or whole with legal structural information. By providing knowledge about the structure of an input string, the number of parse trees can be reduced and a correct parse can be found earlier. A property of the proposed transformation is that it does not generate spurious ambiguous parse trees. This property is non-trivial, as shall be shown later.

In the following section we indicate how an input string can be annotated with structural information by using brackets. In the third section a recipe is given to convert a CFG to a BCFG which can parse the annotated input strings. Features of the recipe are discussed in section 4, before the conclusion is given in section 5.

1.2 Annotating an input string with structural information

In previous studies (e.g. McNaughton (1967) and Knuth (1967)) structural information of the input string was added by placing a pair of brackets around each constituent (a chunk) of the input string. Our method also allows partly annotated (incomplete) input strings:

```
( The cat ) ( has caught ( a mouse ) ).
```

Three chunks can be distinguished here: `The cat`, `a mouse` and `has caught a mouse`.

It is also possible that knowledge about the type of the chunk is present (for example a noun phrase or a verb phrase, `NP` or `VP`). It should be possible to store this information, since more information about the structure may reduce the number of possible parse trees. In our model we will indicate the type of a chunk by subscripting the brackets of the chunk with this type. This way differs from Ginsburg and Harrison (1967), in which each production rule contains a pair of uniquely indexed brackets ($A \rightarrow [_1\ \ldots\ ]_1$). Another difference is that in our annotation method incomplete input strings are possible. Note that each bracket in a pair of brackets must have the same subscript:

```
The cat (VP has caught (NP a mouse )NP )VP.
```

Besides allowing incomplete input strings, our method also allows for inconsistent input strings. In this case the number of opening brackets does not equal the number of closing brackets. We will use square brackets to indicate the start and/or the end of a chunk individually (`[` and `]`). In this case information about the structure of an input string is also present - although more limited than in the other case. Note that it is possible that an opening square bracket and a closing square bracket *may* form a chunk, as is shown in the following inconsistent input string:

```
The cat [VP has caught [NP a mouse ].
```

In this case it is left undecided which pair of brackets form a chunk. When certainty exists about the beginning and end of the same chunk, it is better to use the round brackets to indicate all knowledge about the structure.

Three methods can be used to indicate knowledge about the structure of an input string:

- Define a complete chunk: (...)
- Define the start and/or end of a chunk: [and]
- Define type A of a chunk: $[_A$, $]_A$, $(_A$ ... $)_A$

The three methods can be combined as can be seen in the examples below:

$$(\ \text{The cat}\ )\ [_{\text{VP}}\ \text{has caught}\ (_{\text{NP}}\ \text{a mouse}\ )_{\text{NP}}\ .$$

$$[_{\text{VP}}\ (_{\text{NP}}\ \text{The mouse}\ )_{\text{NP}}\ \text{walked through}\ [_{\text{NP}}\ \text{the barn}\ ]\ .$$

It was mentioned earlier that each single bracket in a pair of typified brackets should have the same subscript. It is also possible to subscript only one of the brackets, after which (in a separate processing step) both brackets should be given the same subscript. For this method it is necessary to find out which round brackets form a pair. This can be realised in a straightforward way. A pair of round brackets is identified by matching an opening round bracket to the nearest closing round bracket, in such a way that the number of opening round brackets equals the number of closing round brackets between them.

In this study, we have restricted ourselves to allow only structural information for non-empty chunks. This decision will be treated in more detail in section 4.

1.3 Converting a CFG to a BCFG

In the previous section we indicated how structural information can be added to an input string by using brackets and subscripts. The following step is to convert the original CFG to a grammar which can also parse the round and square brackets (a BCFG). Note that if the structure symbols are in the set of terminals of the original CFG, other structure symbols should be chosen.

1.3.1 Ambiguity problems

A first approach to create the new grammar is to generate for each production rule in the CFG, $A \rightarrow \ldots$, the following 11 production rules in the BCFG G_f:

$$
\begin{array}{lllll}
A & \rightarrow & & \ldots & \\
A & \rightarrow & & \ldots &] \\
A & \rightarrow & & \ldots &]_A \\
A & \rightarrow & [& \ldots & \\
A & \rightarrow & [& \ldots &] \\
A & \rightarrow & [& \ldots &]_A \\
A & \rightarrow & [_A & \ldots & \\
A & \rightarrow & [_A & \ldots &] \\
A & \rightarrow & [_A & \ldots &]_A \\
A & \rightarrow & (& \ldots &) \\
A & \rightarrow & (_A & \ldots &)_A
\end{array}
$$

In this way all possible configurations of brackets are represented and no parse trees will be generated which do not comply with the indicated structure. The following example illustrates this (the start symbol is A):

$$
\begin{array}{lll}
A & \rightarrow & B\ C \\
A & \rightarrow & C\ D \\
B & \rightarrow & \mathsf{a} \\
C & \rightarrow & \mathsf{a} \\
D & \rightarrow & \mathsf{a}
\end{array}
$$

The input string aa can be parsed in two ways with this grammar:

- $A \Rightarrow B\ C \stackrel{*}{\Rightarrow} \mathsf{aa}$
- $A \Rightarrow C\ D \stackrel{*}{\Rightarrow} \mathsf{aa}$

If it is known in advance that the second a is of type D, this can be indicated by annotating the input string in the following way: $\mathsf{a}\ [_D\ \mathsf{a}$. To parse this input string, the following generated production rules of G_f are relevant (the other production rules are left out for simplicity):

$$
\begin{array}{lllll}
A & \rightarrow & & B\ C & \\
A & \rightarrow & & C\ D & \\
B & \rightarrow & & \mathsf{a} & \\
C & \rightarrow & & \mathsf{a} & \\
D & \rightarrow & & \mathsf{a} & \\
D & \rightarrow & [_D & \mathsf{a} &
\end{array}
$$

The structure symbol can only be matched in the final production rule, therefore the annotated input string can be parsed in one way only: $A \Rightarrow C\ D \stackrel{*}{\Rightarrow} \mathsf{a}\ [_D\ \mathsf{a}$.

By applying this naive conversion to generate the BCFG, it is possible that for a given annotated input string a large number of spurious ambiguous parse trees are generated, which map - when the brackets are removed - on the same parse tree according to the original grammar. This is illustrated with the following CFG:

$$\begin{array}{lcl} A & \rightarrow & A\ \mathtt{a} \\ A & \rightarrow & \mathtt{a} \end{array}$$

If the input string is $[_A\ \mathtt{aa}$, the following generated production rules of G_f are relevant:

$$\begin{array}{lcll} A & \rightarrow & & A\ \mathtt{a} \\ A & \rightarrow & [_A & A\ \mathtt{a} \\ A & \rightarrow & & \mathtt{a} \\ A & \rightarrow & [_A & \mathtt{a} \end{array}$$

Figure 1 shows that the input string $[_A\ \mathtt{aa}$ can be parsed in two ways with the BCFG, while only one parse tree exists for the unannotated input string in the original grammar. According to G_f more parse trees are generated than according to the original CFG, which is of course an undesired property.

The general problem is that G_f does not fix in which production rule the square bracket ($[$ or $]$) is matched. This problem can occur with typified brackets when a production rule of the same type as the bracket is traversed multiple times before the terminal is reached. For example: $A \Rightarrow B \Rightarrow C \Rightarrow A \Rightarrow \mathtt{t}$ or $A \Rightarrow B \Rightarrow C\ \alpha \Rightarrow A\ \alpha\ \beta \Rightarrow \mathtt{t}\ \alpha\ \beta$. If the type of the (opening) bracket equals A, the bracket can be matched at the first or at the final production rule and multiple spurious ambiguous parse trees are generated. If the brackets are not typified this problem occurs when multiple production rules (non-terminals) are traversed before the terminal is reached. For example: $A \Rightarrow B \Rightarrow C \Rightarrow \mathtt{t}$ or $A \Rightarrow \alpha\ B \Rightarrow \alpha\ \beta\ C \Rightarrow \alpha\ \beta\ \mathtt{t}$. Because the (closing) bracket can be matched at every non-terminal, again multiple spurious ambiguous parse trees are generated.

If round brackets are used, the ambiguity problem occurs when unit rules are traversed. If the grammar is converted to Chomsky Normal Form, the problem with regard to the round brackets is solved, however the problem with the square brackets still remains.

1.3.2 Matching brackets as soon as possible

The problem of the previous approach, was the existence of ambiguity in the moment of matching the brackets. A solution for this problem is to define exactly when a bracket should be matched. In the following we will give a conversion of a CFG to a BCFG which enforces that brackets will be matched as soon as possible.

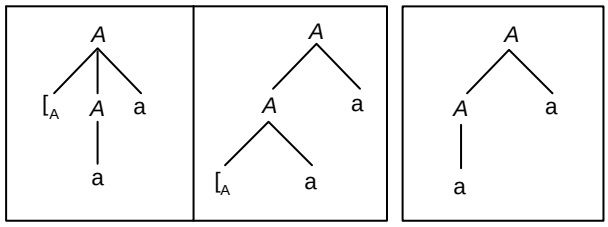

FIGURE 1 *Parse trees for input* $[_A$ aa *and* aa *(left: BCFG, right: CFG)*

Short introduction to the method

In the following method a large number of new production rules in the BCFG (G) are generated for each production rule in the CFG, based on the possible structure symbols. By using two variables (s_0 and $s_0{}'$) for each production rule in G, the structure symbol expected at the start (s_0) and at the end ($s_0{}'$) of the current input string are stored. Because it is not always possible to match a certain structure symbol in a production rule, it is necessary to store for each non-terminal in the right-hand side of the production the structure symbols with which these may start and end. This is done by assigning to each non-terminal in the right-hand side of the production rule two variables, which therefore map to the left side of the generated production rules. By using these variables it is enforced that if a matchable square bracket is not matched in a production rule, it can also not be matched in a later stage in the same parse tree. If round brackets are not matched, they can not be matched in a later stage as long as unit rules are encountered. A more in-depth explanation will be given after the conversion scheme is introduced.

When a specific bracket is expected as a start or end symbol of the current input string, this is indicated by setting the value of the variable (s_0 or $s_0{}'$) equal to this bracket. If no structure symbol may be matched, the symbol ε is used to indicate this.

1.3.3 Conversion scheme CFG → BCFG

The following definitions are used with the conversion:

- N: the set of all non-terminals in the CFG
- T: the set of all terminals in the CFG
- $\Omega_{[} = \{[_A \colon A \in (N \cup \varepsilon)\}$
- $\Omega_{]} = \{]_A \colon A \in (N \cup \varepsilon)\}$
- $\Omega_{(} = \{(_A \colon A \in (N \cup \varepsilon)\}$
- $\Omega_{)} = \{)_A \colon A \in (N \cup \varepsilon)\}$
- $\Omega_b = \Omega_{[} \cup \Omega_{(} \cup \varepsilon$
- $\Omega_{b'} = \Omega_{]} \cup \Omega_{)} \cup \varepsilon$

Note that T must be different from the introduced structure symbols. If this is not the case, different structure symbols must be used.

In the BCFG, we add for each production rule of the CFG

$$A \rightarrow X_1 \ldots X_m$$

with $X_i \in \{N \cup T\}$, new production rules

$$A(s_0, {s'}_0) \rightarrow Y\ {X_1}'\ldots {X_m}'\ Y'$$

with

- $(s_0, {s_0}') \in \Omega_b \times \Omega_{b'}$
- $X_i \in T \Rightarrow {X_i}' = X_i$
- $X_i \in N \Rightarrow {X_i}' = X_i(s_i, {s_i}'), (s_i, {s_i}') \in \Omega_b \times \Omega_{b'}$
- $Y \in \{(, [, [_A, (_A, \varepsilon\}$
- $Y' \in \{),],]_A,)_A, \varepsilon\}$

Where exactly one condition of 1. and one condition of 2. must hold.

For instance, to make sure an opening square bracket is matched at the first possibility, condition 1a. is used. Condition 1a. indicates that when an opening square bracket has no type or a type corresponding to the current production rule, it must be matched because Y is also equal to this bracket (see condition 2a. for the closing square bracket case). Alternatively, if no structure symbol may be matched at the start of a sub-string, condition 1h. is used. Condition 1h. indicates that when no structure symbol may be matched at the start of a certain sub-string (s_0 equals ε), this will hold because Y and s_1 must also equal ε (see condition 2h. for the same case at the end of a sub-string). A detailed explanation of all conditions is given in paragraph 3.4.

1. (a) $s_0 = [_t \wedge t \in \{A, \varepsilon\} \wedge Y = s_0$
   (b) $s_0 \in \Omega_[\setminus \{[_A, [\} \wedge X_1 \in N \wedge Y = \varepsilon \wedge s_1 = s_0$
   (c) $s_0 = (_t \wedge {s_0}' =)_t \wedge t \in \{A, \varepsilon\} \wedge Y = s_0 \wedge Y' = {s_0}'$
   (d) $s_0 = (_t \wedge {s_0}' =)_t \wedge t \in \{A, \varepsilon\} \wedge X_1 \in N \wedge m > 1 \wedge Y = \varepsilon \wedge s_1 = s_0$
   (e) $s_0 \in \Omega_(\setminus \{(_A, (\} \wedge X_1 \in N \wedge Y = \varepsilon \wedge s_1 = s_0$
   (f) $s_0 = (_t \wedge {s_0}' \neq)_t \wedge t \in \{A, \varepsilon\} \wedge X_1 \in N \wedge Y = \varepsilon \wedge s_1 = s_0$
   (g) $s_0 = \varepsilon \wedge X_1 \in T \wedge Y = \varepsilon$
   (h) $s_0 = \varepsilon \wedge X_1 \in N \wedge Y = \varepsilon \wedge s_1 = \varepsilon$
2. (a) ${s_0}' =]_t \wedge t \in \{A, \varepsilon\} \wedge Y' = {s'}_0$
   (b) ${s_0}' \in \Omega_] \setminus \{]_A,]\} \wedge X_m \in N \wedge Y' = \varepsilon \wedge {s_m}' = {s_0}'$
   (c) ${s_0}' =)_t \wedge s_0 = (_t \wedge t \in \{A, \varepsilon\} \wedge Y' = {s_0}' \wedge Y = s_0$
   (d) ${s_0}' =)_t \wedge s_0 = (_t \wedge t \in \{A, \varepsilon\} \wedge X_m \in N \wedge m > 1 \wedge Y' = \varepsilon \wedge {s_m}' = {s_0}'$
   (e) ${s_0}' \in \Omega_) \setminus \{)_A,)\} \wedge X_m \in N \wedge Y' = \varepsilon \wedge {s_m}' = {s_0}'$
   (f) ${s_0}' =)_t \wedge s_0 \neq (_t \wedge t \in \{A, \varepsilon\} \wedge X_m \in N \wedge Y' = \varepsilon \wedge {s_m}' = {s_0}'$
   (g) ${s_0}' = \varepsilon \wedge X_m \in T \wedge Y' = \varepsilon$
   (h) ${s_0}' = \varepsilon \wedge X_m \in N \wedge Y' = \varepsilon \wedge {s_m}' = \varepsilon$

An ε-production rule ($A \rightarrow \varepsilon$) in the CFG is converted to $A(\varepsilon, \varepsilon) \rightarrow \varepsilon$ in the BCFG. As mentioned earlier, we only allow structural information for non-empty chunks.

1.3.4 Explanation of the conversion scheme

For each production rule A of the CFG a number of new production rules are generated in the BCFG G (because of $(s_0, s_0{}') \in \Omega_b \times \Omega_{b'}$). For example, for a non-ε-production rule of a CFG:

$$A \quad \rightarrow \quad \ldots$$

the conversion to G will generate at least 11 new production rules:

$$\begin{array}{lcll}
A(\varepsilon\,,\,\varepsilon) & \rightarrow & & \ldots \\
A(\varepsilon\,,\,\texttt{]}) & \rightarrow & & \ldots\ \texttt{]} \\
A(\varepsilon\,,\,\texttt{]}_A) & \rightarrow & & \ldots\ \texttt{]}_A \\
A(\texttt{[}\,,\,\varepsilon) & \rightarrow & \texttt{[} & \ldots \\
A(\texttt{[}\,,\,\texttt{]}) & \rightarrow & \texttt{[} & \ldots\ \texttt{]} \\
A(\texttt{[}\,,\,\texttt{]}_A) & \rightarrow & \texttt{[} & \ldots\ \texttt{]}_A \\
A(\texttt{[}_A\,,\,\varepsilon) & \rightarrow & \texttt{[}_A & \ldots \\
A(\texttt{[}_A\,,\,\texttt{]}) & \rightarrow & \texttt{[}_A & \ldots\ \texttt{]} \\
A(\texttt{[}_A\,,\,\texttt{]}_A) & \rightarrow & \texttt{[}_A & \ldots\ \texttt{]}_A \\
A(\texttt{(}\,,\,\texttt{)}) & \rightarrow & \texttt{(} & \ldots\ \texttt{)} \\
A(\texttt{(}_A\,,\,\texttt{)}_A) & \rightarrow & \texttt{(}_A & \ldots\ \texttt{)}_A
\end{array}$$

Because non-terminals may exist in the right-hand side of the production rule A, it is possible that there are more production rules generated. This is discussed later.

The large number of generated production rules is necessary, because there must exist a production rule for each structure symbol in which it can be matched. If more non-terminals are present in the CFG, the number of structure symbols also increases (and this results in a larger grammar). An analysis of the number of generated production rules, based on the original production rules, the number of terminals and non-terminals in the CFG is given in a later section.

The conversion scheme enforces that terminals and non-terminals (X_i) remain in the same place in the generated production rule $A(s_0, s_0{}')$ as in the original production rule A.

The variables s_0 and $s_0{}'$ indicate which structure symbols are expected at the start and the end of the current input string. The variables Y and Y' indicate which structure symbols must be matched at the start and at the end of the current production rule.

As discussed earlier, ambiguity with respect to matching the brackets can occur with square brackets and round brackets in combination with unit-rules. This ambiguity is prevented by matching the structure symbols as soon as this is possible. The values of Y and Y' will therefore correspond when this is possible with s_0 and $s_0{}'$.

In the next two paragraphs the influence of s_0 and $s_0{}'$ on Y and s_1 will be discussed. The situation for Y' and $s_m{}'$ is analogous, instead of the conditions of 1. the conditions of 2. will be used. The relevant conditions of the conversion scheme are mentioned at the end of each paragraph.

The influence of s_0 and $s_0{}'$ on Y

When a square bracket without a type is expected ($s_0 = [$), this symbol can be matched in every production rule and thus the value of Y must equal s_0. This is also the case if a typified square bracket is expected with a type corresponding with the current production rule, $s_0 = [_A$ (**1a**).

When a pair of round brackets without a type is expected ($s_0 = ($ and $s_0{}' =)$), these structure symbols can be matched in every production rule. If the production rule is a unit-rule or starts and/or ends with a terminal, the values of Y and Y' must equal the values of s_0 and $s_0{}'$. If this is not the case, the values of Y and Y' must equal the values of s_0 and $s_0{}'$ or must both be equal to ε. Since the values of s_0 and $s_0{}'$ do not have to apply to the same chunk and can be matched later, the values of Y and Y' can also be equal to ε. The same arguments can be applied for a situation in which a pair of typified round brackets is expected with a type corresponding to the current production rule, $s_0 = (_A$ and $s_0{}' =)_A$ (**1c,d**).

If no structure symbol can be matched, $s_0 = \varepsilon$, Y is left out (**1g,h**).

Finally, if a typified bracket is expected with a type not corresponding to the current production rule, it is not possible to match this structure symbol in the current production rule. This is also the case if a matchable opening round bracket is expected without the matchable closing round bracket. If the right-hand side of the production rule does not start with a terminal, the value of Y must equal ε (**1b,e,f**). In the other case no production rule is generated, because the typified bracket cannot be matched.

The influence of s_0 and $s_0{}'$ on s_1

If no structure symbol can be matched, the current input string w may not start with a structure symbol. If the right-hand side of the current production rule A starts with a non-terminal B, the start of w is parsed with the production rule belonging to B. Since w may not start with a structure symbol, the production rule of B may not start with a structure symbol. Therefore the value of s_1 must be equal to ε (**1h**).

If a typified bracket of a different type than A is expected at the start of w, this structure symbol cannot be matched in the current production rule. This is also the case if a matchable opening round bracket is expected without a matchable closing round bracket. In these cases the value of s_0, like in the previous situation, must be passed on to B ($s_1 = s_0$) where the structure symbol can possibly be matched (**1b,e,f**).

If a pair of round brackets can be matched and the right-hand side of the non-unit production rule starts and ends with a non-terminal, it is also possible to pass on the round

brackets. In that situation s_1 must be equal to s_0. This has to be possible, because s_0 and $s_0{}'$ can apply to different chunks and therefore should be matched later. Only in the previous three situations, the value of s_1 is specified. If a structure symbol is matched in the current production rule, the value of s_1 is free (**1a,c**).

The value of the variable $s_1{}'$ is free if the right-hand side of the production rule does not consist of one element (being a non-terminal). The values of the other variables s_i and $s_i{}'$ for $1 < i < m$ are always free.

Free variables

If the value of one or more variables (s_i and $s_i{}'$) is free, this results in the generation of multiple production rules for the same $A(s_0, s_0{}')$. For every possible combination of variable values s_i and $s_i{}'$ a production rule must exist. This is illustrated by the following production rule (the complete CFG consists of two non-terminals):

$$A \quad \rightarrow \quad B\ \mathsf{b}$$

We limit ourselves to the generated production rules for $A(\texttt{[}_B, \varepsilon)$. This means that at the beginning a typified bracket is expected unequal to the current type ($B \neq A$) and at the end no structure symbol may be present:

$$\begin{array}{lcl}
A(\texttt{[}_B, \varepsilon) & \rightarrow & B(\texttt{[}_B, \varepsilon)\ \mathsf{b} \\
A(\texttt{[}_B, \varepsilon) & \rightarrow & B(\texttt{[}_B, \texttt{)})\ \mathsf{b} \\
A(\texttt{[}_B, \varepsilon) & \rightarrow & B(\texttt{[}_B, \texttt{)}_A)\ \mathsf{b} \\
A(\texttt{[}_B, \varepsilon) & \rightarrow & B(\texttt{[}_B, \texttt{)}_B)\ \mathsf{b} \\
A(\texttt{[}_B, \varepsilon) & \rightarrow & B(\texttt{[}_B, \texttt{]})\ \mathsf{b} \\
A(\texttt{[}_B, \varepsilon) & \rightarrow & B(\texttt{[}_B, \texttt{]}_A)\ \mathsf{b} \\
A(\texttt{[}_B, \varepsilon) & \rightarrow & B(\texttt{[}_B, \texttt{]}_B)\ \mathsf{b}
\end{array}$$

If there are more free variables present (s_i or $s_i{}'$), this results in a significant increase of the number of production rules of G. This will be explained in more detail later.

Several examples of parsing an annotated input string by a BCFG are given in appendix A (downloadable from: http://www.martijnwieling.nl).

1.3.5 Converting generated parse trees

After the annotated input string has been parsed according to the BCFG, the final step is to convert the BCFG parse trees to CFG parse trees. This can be realized very easily by applying the following two steps (this is also illustrated in figure 2):

- Every $A(s_i, s_i{}')$ is replaced by A
- All structure symbols ($Y \neq \varepsilon$) are removed

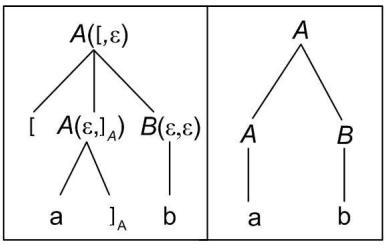

FIGURE 2 *Conversion of generated parse trees (left: BCFG, right: CFG)*

1.3.6 Properties of the BCFG

In this paragraph we show that the BCFG can parse all legally annotated input strings. A legally annotated input string means that there exists a parse tree in the CFG for the unannotated input string, which adheres to the structure indicated by the annotation. We also show that no extra ambiguity is caused by the annotation of the input string with structural information.

The conversion scheme enforces that terminal and non-terminal symbols remain in the same order as in the CFG. The only difference between the CFG and the BCFG is therefore the use of structural information. We therefore will focus on this aspect in the following.

- *Property 1*: The BCFG can parse all input strings which can be constructed with the CFG with the addition of legal structural information

Proof: The conversion scheme stores (by using s_0 and $s_0{}'$) the structure symbols with which the current input string may start and end. Because of $(s_0, s_0{}') \in \Omega_b \times \Omega_{b'}$ all combinations of matching structure symbols are present for every production rule and the current input string may therefore start and end with all possible structure symbol combinations. Because of $(s_i, s_i{}') \in \Omega_b \times \Omega_{b'}$, the non-terminals (parsing sub-strings) on the right-hand side of every production rule may also start and end with all possible structure symbol combinations. The only exception is that s_1 and $s_m{}'$ may depend on s_0 and $s_0{}'$ respectively, but this is only the case when they indicate structure symbols which are expected at the start or end of the current input string (and for this case all possible explanations were possible).

Since a square bracket or a pair of round brackets can be matched only if it does not have a type, or has a type corresponding with the current production rule (see condition a and c), only input strings can be parsed which have a legal annotation.

- *Property 2*: The BCFG does not generate CFG-equivalent parse trees for an input string.

Proof: CFG-equivalence of two BCFG parse trees means that if both BCFG parse trees are converted to CFG parse trees (see the previous paragraph) these parse trees are identical.

Assume there exist two BCFG parse trees for a certain annotated input string which are CFG-equivalent. In that case, it is necessary that a structure symbol is present in different places in the parse tree. This means that it must be possible to ignore a structure symbol when it can be matched first and subsequently match it in a later stage (without parsing terminals in between).

To ignore a matchable opening square bracket, the corresponding variable (s_0) must be equal to ε (see condition g and h). This results in s_1, if it is present, being equal to ε. As a consequence, the value s_0 of the production rule X_1 will also be equal to ε. This process will repeat itself. To parse the input string correctly, a terminal must be matched (see condition g). This shows that it is not possible to ignore a matchable opening square bracket and match it in a later stage, before matching a terminal.

The case for a matchable closing square bracket is identical, with s_0 replaced by $s_0{}'$, s_1 by $s_m{}'$ and X_1 by X_m.

The same arguments (for s_0 **and** $s_0{}'$) hold for a pair of round brackets if the right-hand side of the production rule consists of one non-terminal. If this is not the case (condition d) round brackets can be ignored, but can never be matched again defining the same chunk. No production rules are generated where a single round bracket can be matched.

As we have shown, it is not possible to ignore a matchable structure symbol and match it in a later stage without matching a terminal in between. This contradicts our assumption and we can conclude that there are no CFG-equivalent parse trees generated for a certain annotated input string.

1.3.7 Number of generated production rules

The BCFG will consist of a large number of production rules which depends on the number of non-terminals (N) in the CFG, the number of non-terminals (Z) in the right-hand side of every single production rule and the type of X_1 and X_m (terminal or non-terminal). An ε-production rule in the CFG will only generate a single production rule in the BCFG.

Four other cases can be distinguished:

1. X_1 and X_m are both terminals
2. X_1 is a terminal and X_m is a non-terminal, or vice versa
3. X_1 and X_m are both non-terminals and $m > 1$
4. X_1 is a non-terminal and $m = 1$

When a production rule only consists of terminals, 11 production rules will be generated in the BCFG. In this case no ambiguity problem exists and the same production rules are generated as for G_f (section 3). When Z non-terminals are present in the production rule (not at the start and the end), $2Z$ free variables are present (s_i and $s_i{}'$). Every free variable has $2N + 3$ possible values ($|\Omega_b|$ or $|\Omega_{b'}|$). The number of generated production rules in the BCFG for a production rule in the CFG which starts and ends with a non-terminal is

therefore given by the following formula:

$$C_0 = 11 \cdot (2N + 3)^{2Z} \tag{1.1}$$

The total number of generated production rules in the BCFG for a production rule of the CFG beginning with a terminal and ending with a non-terminal (or vice versa) is given by the following formula[1]:

$$C_1 = (22N + 27) \cdot (2N + 3)^{(2Z-1)} \tag{1.2}$$

For a production rule of the CFG which starts and ends with a non-terminal and $m > 1$, the following formula is used to calculate the number of generated production rules in the BCFG[1]:

$$C_2 = (44N^2 + 108N + 67) \cdot (2N + 3)^{(2Z-2)} \tag{1.3}$$

When a production rule of the CFG consists only of one non-terminal ($m = 1$), the number of production rules in the BCFG is given by the following formula[1]:

$$C_3 = 44N^2 + 108N + 65 \tag{1.4}$$

For C_0, C_1, C_2 and C_3 it is clear that the number of generated production rules equals $O(N^{2Z})$. The total number of generated production rules in the BCFG based on a CFG consisting of

- p production rules where X_1 and X_m are both terminals
- q production rules where X_1 is a terminal and X_m is a non-terminal (or vice versa)
- r production rules where X_1 and X_m are both non-terminals and $m > 1$
- s production rules where X_1 is a non-terminal and $m = 1$
- t ε-production rules

thus equals:

$$|G| = p \cdot C_0 + q \cdot C_1 + r \cdot C_2 + s \cdot C_3 + t$$

If the original CFG is converted to Chomsky Normal Form, the right-hand side of every production rule in the CFG consists of one terminal or two non-terminals. In this case q, s and t equal 0, the value of Z equals 0 for C_0 and the value of Z equals 2 for C_2. The total number of generated production rules G_c then equals:

$$|G_c| = p \cdot C_0 + r \cdot C_2$$

with $C_0 = 11$ and $C_2 = (44N^2 + 108N + 67) \cdot (2N + 3)^2$. The number of generated production rules in G_c thus has a polynomial degree, $O(N^4)$.

If the CFG is not in Chomsky Normal Form, but the highest number of non-terminals in the right-hand side of a production rule of the CFG is known (Z_{max}), the number of generated production rules also has a polynomial degree, $O(N^{2Z_{max}})$.

[1] A precise calculation is given in appendix B (downloadable from: http://www.martijnwieling.nl)

1.4 Discussion

We did not investigate in what way the size of the BCFG influences the time needed to parse an input string. Both the Earley-algorithm and the CYK-algorithm have a time complexity depending on the size of the grammar and therefore will be influenced. However, it is likely that new production rules can be generated on the fly and thus will alleviate the problem.

When it is undesirable to use a BCFG with a large number of production rules, it is also possible to use the ambiguous conversion scheme. After the parse trees have been generated according to this BCFG (G_f with the addition that ε-production rules remain the same and do not get any structure symbols), the parse trees have to be converted to CFG parse trees by removing the structure symbols. In a subsequent sweep duplicate parse trees can be then be removed.

In our study we only allow structural information for non-empty chunks. If structural information is also desired for empty chunks, the conversion scheme cannot be adapted very easily. This is illustrated with the following example. In the production rule $A(s_0, s_0{}') \rightarrow X_1(s_1, s_1{}')\ X_2(s_2, s_2{}')$ the value of X_1 equals ε. A square bracket without a type can be matched in the production rule of X_1 (if $s_0 = [$), but it is also possible to match the square bracket in the production rule X_2 while not matching it in X_1 ($s_1 = \varepsilon$). Since the value of s_0 does not influence the value of s_2, spurious ambiguity can occur here.

1.5 Conclusion

We showed how an input string can be annotated with structural information and subsequently can be parsed with a BCFG. A conversion scheme was given to convert a CFG to a BCFG with the important property that the resulting BCFG does not generate spurious ambiguous parse trees.

If an input string is parsed with a CFG a large number of parse trees can be generated. The number of parse trees can be reduced by annotating the input string with structural information, parsing the annotated input string with the converted CFG (the BCFG) and converting the resulting BCFG parse trees to CFG parse trees. The number of parse trees is only reduced when the CFG contains parse trees for the original input string which do not comply to the indicated structure (these parse trees will not be generated by the BCFG).

References

Ginsburg, S. and Harrison, M. A.(1967), Bracketed context-free languages., *J. Comput. Syst. Sci.* **1**(1), 1–23.

Knuth, D. E.(1967), A characterization of parenthesis languages, *Information and Control* **11**(3), 269–289.

McNaughton, R.(1967), Parenthesis grammars, *Journal of the ACM* **14**(3), 490–500.

Pereira, F. and Schabes, Y.(1992), Inside-outside reestimation from partially bracketed corpora, *Proceedings of the 30th annual meeting on Association for Computational Linguistics*, Association for Computational Linguistics, Morristown, NJ, USA, pp. 128–135.

Ramshaw, L. and Marcus, M.(1995), Text chunking using transformation-based learning, *in* D. Yarovsky and K. Church (eds), *Proceedings of the Third Workshop on Very Large Corpora*, Association for Computational Linguistics, Somerset, New Jersey, pp. 82–94.

Stolcke, A.(1995), An efficient probabilistic context-free parsing algorithm that computes prefix probabilities, *Comput. Linguist.* **21**(2), 165–201.

van der Beek, L., Bouma, G., Malouf, R. and van Noord, G.(2002), The Alpino dependency treebank, *Computational Linguistics in the Netherlands*.

van Noord, G., Schuurman, I. and Vandeghinste, V.(2006), Syntactic annotation of large corpora in STEVIN, *LREC 2006*, Genua.