TextTree Construction for Parser and Treebank Development

Paula S. Newman
newmanp@acm.org

Abstract

TextTrees, introduced in (Newman, 2005), are skeletal representations formed by systematically converting parser output trees into unlabeled indented strings with minimal bracketing. Files of TextTrees can be read rapidly to evaluate the results of parsing long documents, and are easily edited to allow limited-cost treebank development. This paper reviews the TextTree concept, and then describes the implementation of the almost parser- and grammar-independent TextTree generator, as well as auxiliary methods for producing parser review files and inputs to bracket scoring tools. The results of some limited experiments in TextTree usage are also provided.

1 Introduction

The TextTree representation was developed to support a limited-resource effort to build a new hybrid English parser[1]. When the parser reached significant apparent coverage, in terms of numbers of sentences receiving some parse, the need arose to quickly assess the quality of the parses produced, for purposes of detecting coverage gaps, refining analyses, and measurement. But this was hampered by the use of a detailed parser output representation.

The two most common parser-output displays of constituent structure are: (a) multi-line labeled and bracketed strings, with indentation indicating dominance, and (b) 2-dimensional trees. While these displays are indispensable in grammar development, they cannot be scanned quickly. Labels and brackets interfere with reading. And, although relatively flat 2D node + edge trees for short sentences can be grasped at a glance, for long sentences this property must be compromised.

In contrast, for languages with a relatively fixed word order, and a tendency to post-modification, TextTrees capture the dependencies found by a parser in a natural, almost self-explanatory way. For example:

```
They
must have
    a clear delineation
        of
            [roles,
             missions,
             and
             authority].
```

Indented elements are usually right-hand post-modifiers or subordinates of the lexical head of the nearest preceding less-indented line. Brackets are generally used only to delimit coordinations (by [...]), nested clauses (by {…}), and identified multi-words (by |…|).

Reading a TextTree for a correct parse is similar to reading ordinary text, but reading a TextTree for an incorrect parse is jarring. For example, the following TextTree for a 33-word sentence exposes several errors made by the hybrid parser:

```
But
    by |September 2001|,
the executive branch
    of
        [the U.S. government,
         the Congress,
             the news media,
         and
         the American public]
had received
   clear warning
       that
          {Islamist terrorists
           meant
              to kill
                  Americans
                     in high numbers}.
```

[1] The hybrid combines the chunker part of the fast, robust XIP parser (Aït-Mokhtar et al., 2002) with an ATN-style parser operating primarily on the chunks.

TextTrees can be embedded in bulk parser output files with arbitrary surrounding information. Figure 1 shows an excerpt from such a file, containing the TextTree-form results of parsing the roughly 500-sentence "Executive Summary" of the 9/11 Commission Report (2004) by the hybrid parser, with more detailed results for each sentence accessible via links. (Note: the links in Figure 1 are greyed to indicate that they are not operational in the illustration.)

Such files can also be edited efficiently to produce limited-function treebanks, because the needed modifications are easy to identify, labels are unnecessary, and little attention to bracketing is required. Edited and unedited TextTree files can then be mapped into files containing fully bracketed strings (although bracketed differently than the original parse trees), and compared by bracket scoring methods derived from Black et al (1991).

Section 2 below examines the problems presented by detailed parse trees for late-stage parser development activities in more detail. Section 3 describes the inputs to and outputs from the TextTree generator, and Section 4 the generator implementation. Section 5 discusses the use of the TextTree generator in producing TextTree files for parser output review and TextTreebank construction, and the use of TextTreebanks in parser measurement. The results of some limited experiments in TextTree file use are provided in Section 6. Section 7 discusses related work and Section 8 explores some directions for further exploitation of TextTrees.

6 (3) We have come together with a unity of purpose because our nation demands it. best more chunks

```
    We
    have come together
        with a unity
            of purpose
        because
            {our nation
             demands
                 it}.
```

7 (15) September 11 , 2001 , was a day of unprecedented shock and suffering in the history of the United States . best more chunks

```
    |September 11 , 2001|,
    was
        a day
            of
                [unprecedented shock
                 and
                 suffering]
                     in the history
                         of the United States.
```

8 (1) The nation was unprepared . best more chunks

```
    The nation
    was
        unprepared.
```

Figure 1. A TextTree file excerpt

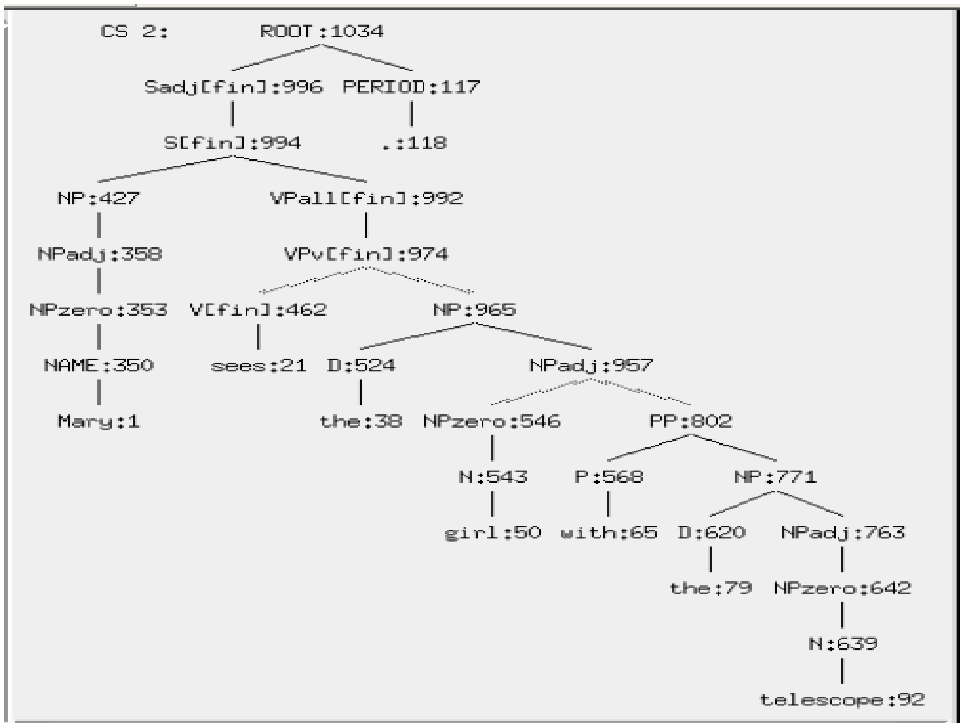


Figure 2. An LFG c-structure

2 Problems of Detailed Parse Trees

This section examines the readability problems posed by conventional parse tree representations, and their implications for parser development activities.

As noted above, parse trees are usually displayed using either 2-dimensional trees or fully bracketed strings. Two dimensional trees are intended to provide a clear understanding of structure. Yet because of the level of detail involved in many grammars, and the problem of dealing with long sentences, they often fail in this regard. Figure 2 illustrates an LFG c-structure, reproduced from King et al. (2000), for the 7 word sentence "Mary sees the girl with the telescope." Similar structures would be obtained from parsers using other popular grammatical paradigms. The amount of detail created by the deep category nesting makes it difficult to grasp the structure by casual inspection, and the tree in this case is actually wider than the sentence.

Grammar-specific transformations have been used in the LKB system to simplify displays (Oepen et al., 2002). But there are no truly satisfactory approaches for dealing with the problem of tree width, that is, for presenting 2D trees so as to provide an "at-a-glance" appreciation of structure for sentences longer than 25 words, which are very prevalent.[2] The methods currently in use or suggested either obscure the structure or require additional actions by the reader. Allowing trees to be as wide as the sentences requires horizontal scrolling. Collapsing some subtrees into single nodes (wrapping represented token sequences under those nodes) requires repeated expansions to view the entire parse. Using more of the display space by overlapping subtrees vertically interferes with comprehension because it obscures dominance and sequence. In Figure 2, for example, the period at the end of the sequence is the second constituent at the top. For a longer sentence, a coordinated constituent might be placed here as well. Such unpredictable arrangements impede reading, because the reader does not know where to look

[2] Casual records of parser results for many English non-fiction documents suggest an average of about 20 words per sentence, with a standard deviation of about 11.

```
S (NP-SBJ Stokely)
  (VP says
     (SBAR  0
        (S (NP-SBJ stores)
           (VP revive
              (NP (NP specials)
                  (PP like
                      (NP (NP three cans)
                          (PP of
                              (NP peas))
                          (PP for
                              (NP 99 cents )))))))))
```

Figure 3. A Penn Treebank Tree

The other conventional parse tree representation is as a fully-bracketed string, usually including category labels and, for display purposes, using indentation to indicate dominance. Figure 3 shows such a tree, drawn from a Penn Treebank annotation guide (Bies et al., 1995), shown with somewhat narrower indentation than the original. Even though the tree is in the relatively flat form developed for use by annotators, the brackets, labels, and depth of nesting combine to prohibit rapid scanning.

However, it should be noted that this format is the source of the TextTree concept. Because by eliminating labels, null elements, and most brackets, and further flattening, the more readable TextTree form emerges:

```
Stokely
says
    {stores
     revive
         specials
             like three cans
                 of peas
                 for 99 cents}
```

2.1 Implications

The conventional parse tree representations discussed above can be a bottleneck in parser development, most importantly in checking parser accuracy with respect to current focus areas and in extending coverage to new genres or domains. For these purposes, one would like to review the results of analyzing collections of relatively long (100+ sentence) documents. But unless this can be done quickly, a parser developer or grammar writer is tempted to rely on statistics with respect to the number of sentences given a parse, and declare victory if a high rate of parsing is reported.

Another approach to assessing parser quality is to rely on treebank-based bracket scoring measures, if treebanks are available in the particular genre. This can also can be a pitfall, as bracket scores tend to be unconsciously interpreted as measures of full-sentence parse correctness, which they are not.

On the other hand, treebank-based measurements can play a useful secondary role in evaluating parser development progress and in comparing parsers. But if insufficient treebanked material is available for the relevant domains and/or genres, a custom treebank must be developed. This process generally consists of two phases: (a) a bootstrapping phase in which an existing parser is applied to the corpus to produce either a single "best" tree, or multiple alternative trees, for each sentence, and then (b) a second phase in which annotators approve or edit a given parse tree, select among alternative trees, or manually create a full parse tree when no parse exists. All of the second phase alternatives are difficult given conventional parse-tree representations. For example, experiments by Marcus et al. (1993) associated with the Penn Treebank indicated that for the first annotation pass "At an average rate of 750 words per hour, a team of five part-time annotators …", i.e., a bit more than a page of this text per hour. Aids to selecting among alternative 2D trees can be given in the form of differentiating features (Oepen et al., 2002), but their effectiveness in helping to select among large trees differing in attachment choices is not clear.

Another activity impeded by conventional parse tree representations is regression testing. As a grammar increases in size, it is advisable to frequently re-apply the grammar to a large test corpus to determine whether recent changes have had negative effects on accuracy. While the existence of differences in output can be detected by automatic means, one would like to assess the differences by quickly comparing the divergent parses, which is difficult to do using detailed parse displays for long sentences.

Finally, an activity that is rarely discussed but is becoming increasingly important is providing comprehensible parser demonstrations. A syntactic parser is not an end-in-itself, but a building block in larger NLP research and development efforts. The criteria for selecting a parser for a project include both the kind of information provided by the parser and parser accuracy. However, current parser output representations are not geared to allowing potential users to quickly assess accuracy with respect to document types of interest.

3 The TextTree Generator: Externals

TextTrees are generated by an essentially grammar-independent processor from a combination of

(a) parser output trees that have been put into a standard form that retains the grammar-specific category labels and constituents, but whose details conform to the requirements of a parser-independent tool interface, and,

(b) a set of grammar-specific <category, directive> pairs, e.g., "<ROOT, Align>". Each pair specifies a default formatting for constituents in the category, specifically whether their sub-constituents are to be aligned vertically, or concatenated relative to a marked point, with subsequent children indented. These defaults are used in the absence of overriding factors such as coordination.

The directives used are very simple ones, because the simpler the formatting rules, the more likely it is that outputs can be accurately checked for correctness, and edited conveniently

It is assumed that the parser output either includes conventional parse trees, or that such trees can be derived from the output. This assumption covers many grammatical approaches, such as CG, HPSG, LFG, and TAG.

The logical configuration implied by these inputs is shown in Figure 4. It includes a parser-specific adaptor to convert parser-output trees into a standard form. The adaptor need not be very large; for the hybrid parser it consists of about 75 lines of Java code.

The subsections below discuss the standard ParseTree form, the directives that have been identified to date and their formatting effects, and the treatment of coordination.

3.1 Standard ParseTrees

A standard ParseTree consists of recursively

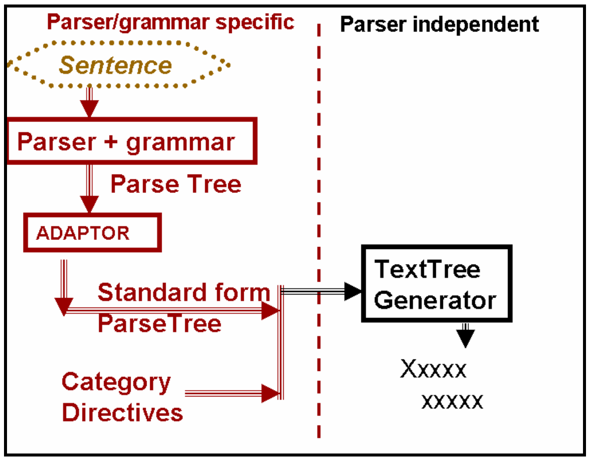


Figure 4: Logical Configuration.

nested subtrees, each representing a constituent C, and each indicating:

- A grammar-specific category label for C.
- Whether C should be considered the head of its immediately containing constituent P for formatting purposes. Generally, this is the case if C is, or dominates, the lexical head of P, but might differ from that to obtain some desired formatting effects. As heads are identified by most parsers, this marker can usually be set by parser-specific rules embedded in the adaptor.
- Whether C is a *coord_dom*, that is, whether it immediately dominates the participants in a coordination.
- If C is a leaf, the associated token, and
- (optionally) whether C is a multi-word.

3.2 Formatting Directives

To generate TextTrees for a particular grammar, a <category, directive> pair is provided for each category that will appear in a ParseTree for the grammar. The directive specifies how to format the children of constituents in the category in the absence of lengthy coordinations.

The definitions of the directives make use of one additional locator for a constituent, its *real_head*. While we generally want the directives to format constituents relative to their lexical heads, in some grammars, those heads are deeply nested. For example, a tree for an NP might be generated by rules such as:

NPz => DET <u>NPy</u>
NPy => ADJ* <u>NPx</u>
NPx => <u>NOUN</u> PP*

where each of the underscored terms are *heads*, but the *real_head* of NPz is the head NOUN of NPx. More generally, the *real_head* of a constituent C is the first constituent found by following the *head*-marked descendants of C downward until either (a) a leaf or headless constituent is reached, or (b) a post-modified *head*-marked constituent is found.

Using this definition, the current formatting directives are as follows:

Align: Align specifies that all children of the constituent are vertically aligned. Thus, for example, a directive <ROOT, Align>, would cause a constituent generated by a rule "ROOT => NP, VP" to be formatted as:

```
formatted NP
formatted VP
```

ConcatHead: ConcatHead concatenates the tokens of a constituent (with separating blanks) up to and including its *real_head* (as defined above), and indents and aligns the post-modifiers of the *real_head*. For example, given the directive "<NP, ConcatHead>", a constituent produced by the rules:

```
NPz => PreMod* NPx
NPx => NOUN PostMods*
```

would be formatted as:

```
All-words-in-PreMod* NOUN
             Formatted PostMod1
             Formatted PostMod2
```

ConcatCompHead: ConcatCompHead concatenates everything up to and including the *real_head*, and concatenates with that the results of a ConcatHead of the first post-modifier of the *real_head* (if any).

This directive is motivated by rules such as "PP => P NP", where the desired formatting groups the head with words up to and including the lexical head of the single complement, e.g.,

```
of the man
    in the moon
```

ConcatSimpleComp: ConcatSimpleComp concatenates material up to and including the *real_head* and, if first post-modifying constituent is a simple token, concatenates that as well, and then aligns and indents any further post-modifiers of the *real_head*. It thus formats noun phrases for languages that routinely use simple adjective post-modifiers. For example (Sp.):

```
La casa blanca
      que ...
```

ConcatPreHead: This directive concatenates material, if any, before the *real_head*, and then if such material exists, increases the indent level. It then aligns the following component. The directive is intended for formatting clauses that begin with subordinating conjunctions, complementizers, or relative pronouns, where the grammar has identified the following sentential component as the head, but it is more readable to indent that head under the initial constituent. In practice, in such cases it is easier to just alter the *head* marker within the adaptor.

3.3 Treatment of Coordination

The directives listed in the previous subsection are defaults that specify the handling of categories in the absence of coordination. A set of coordinated constituents are always indicated by surrounding square brackets ([ ]). If the coordination occurs within a requested concatenation, then if the width of coordination is less than a predesignated size, the non-token constituents of the coordination are bracketed, as in:

```
[{Old men} and women]
    in the park.
```

However, if the width of the coordination exceeds that size, the concatenation is converted to an alignment to avoid line wrap, for example,

```
He
gave
    sizeable donations
        to
            [the church,
             the school,
             and
             the new museum
                 of art.]
```

4 TextTree Generator: Implementation

TextTrees are produced in two steps. First, a ParseTree is processed to form one or more InternalTextTrees (ITTs), which are then mapped into an external TextTree. Most of the work is accomplished in the first step; the use of a second step allows the first to focus on logical structure, independent of many details of indentation, linebreaks, and punctuation.

We begin by describing ITTs and their relationship to external TextTrees, to motivate the description of ITT formation. We then describe the mapping from ParseTrees to ITTs.

4.1 Transforming ITTs to Strings

Figure 5 shows a simple ITT and its associated external TextTree. The node labels of an ITT are called the "headparts" of their associated subtrees. Headparts may be null, simple strings, or references to other ITTs.

If the headpart of a subtree is null, like that of the outermost subtree of Figure 5, the external TextTrees for its children are aligned vertically at the current level of indentation. Also, if the subtree is not outermost, the aligned sequence is bracketed.

However, if the headpart of a subtree is a simple string, as in the other subtrees of Figure 5 ITT, that string is printed at the current level of indentation, and the external TextTrees for its children, if any, are aligned vertically at the next level of indentation.

The headpart of a subtree may also reference another ITT, as illustrated in Figure 6. Such a reference headpart signals the bracketed inclusion of the referenced tree, which has a null headpart, at the current indentation level. This permits the entire referenced tree to be post-modified. The brackets used depend on a feature associated with the referenced tree, and are either [ ], if the referenced tree represents a coordination, or { } otherwise.

Pseudo-code for the ITT2String function that produces external TextTrees from ITTs is shown in Figure 7. The code omits the treatment of non-bracketing punctuation; in general, care is taken to prefix or suffix punctuation to tokens appropriately.

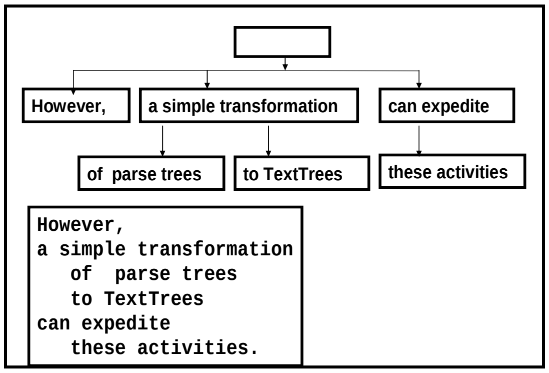
4.2 Transforming ParseTrees to ITTs

To transform ParseTrees into ITTs, subtrees are processed recursively top-down using the function BuildITT of Figure 8. Its arguments are an input subtree *p*, and a directive *override*, which may be null. It returns an ITT for *p*.

BuildITT sets the operational directive *d* as either the *override* argument, if that is non-null, or to the default directive for the category of *p*.

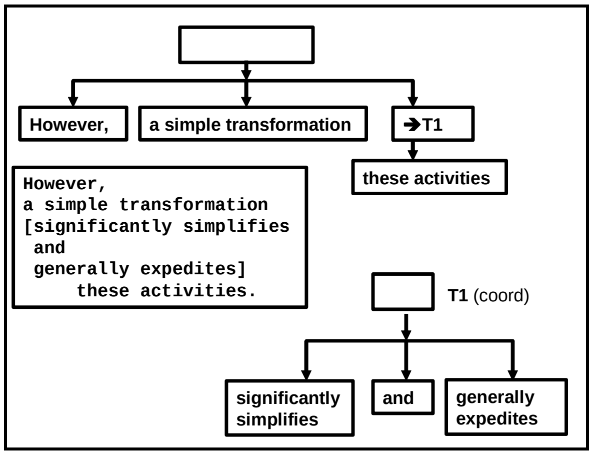
Then, if *d* is "Align", the result ITT has a null headpart and, if *p* is a *coord_dom*, the isCoord property of the ITT is set. The children of the result ITT are the ITTs of *p*'s children.

However, if *d* specifies that an initial sequence of *p* is to be concatenated, BuildITT uses the recursive function Catenate (Figure 9) to obtain the initial sequence if possible.



```
However,
a simple transformation
   of  parse trees
   to TextTrees
can expedite
   these activities.
```

Figure 5: Simple InternalTextTree.



```
However,
a simple transformation
[significantly simplifies
 and
 generally expedites]
     these activities.
```

Figure 6: ITT with long coordination

```
Function ITT2String(ITT s, String indent)
                    returns String
// indent is a string of blanks, eol is end-of-line
Set ls to null
If s has a null headpart, set nextIndent to indent
      + 1 blank  (adds space for [ or { bracket )
Otherwise  set nextIndent to indent + N blanks,
      where N is the constant indent increment
If s has a headpart reference
      set nextIndent to nextIndent + 1 blank

If s has children, set ls to the lines produced
      by ITT2String (ci, nextIndent)
            for each child ci of s

If s has a headpart string hs
      Return the concatenation of
            indent, hs, eol, ls
Else if s has a headpart reference to an ITT s2
      Return the concatenation of
            ITT2String(s2, indent), ls
Else  (s has a null headpart )
      Remove initial & trailing whitespace from ls
      If s is a coordination, set pfx to [ and sfx to ]
      Else set pfx to {and sfx to }
      Return the concatenation of
            nextIndent – 1 blank, pfx, ls, sfx, eol
```

Figure 7. The ITT2String Function

```
Function Catenate (ParseTree p,
      ParseTree r)  returns <Code, String>
1. Set result = null, code = incomplete.
     If p is a leaf, result = the token of p.
2. for each child ci of p,
    while $code \neq$ complete:
     If ci = r, set code = complete.
     Set <ccode, cstring> to result of
            Catenate(ci, r)
      If (ccode = failed) return <failed, null>
      If p is a coord_dom & cstring not 1 word
            // indicate coordinated within concat
            Suffix "{cstring}" to result.
     Otherwise suffix "cstring" to result
     If ccode = complete, set code = complete
3. Finally,
     If p = r, set code = complete.
     if p is a coord_dom and
        the length of  result  > LONG_CONST,
        return <failed, null>.
    Else if p is a coord_dom
          Return <code, "[result]">
    Else if p is a multi-word,
          return <code, "|result|">
     Otherwise return <code, result>
```

Figure 9. The Catenate Function

```
Function BuildITT(ParseTree p,  Directive override)  returns ITT
Set  d to override if non-null, otherwise to the default category directive for p.
1. If p is a leaf or a multiword:
    -   return an ITT whose headline concatenates the tokens spanned by p, and which has no children.
        If p is a multiword, bracket the headline by |.
2. Else if  d is Align
        return an ITT with a null headpart, and children built by invoking BuildITT(ci, null) for each
        child ci of p. If p is a coord_dom, indicate that the ITT isCoord.
3. Otherwise concatenate:
a) Find the nested subtree s that contains the rightmost element r to be concatenated according to d.
b)  Set the pair <ccode, cstring> to the result of Catenate (p, r).
c) If $ccode \neq$ "failed",   return ITT with headpart = cstring, and children formed by BuildITT(ci, null)
    for each child ci of s after r.  .
d) Otherwise return a directive-dependent tree aligning the contained coordination, for example:
   For ConcatHead:  i. Let p'  be like p but without the right siblings of the real_head of  p
                    ii. Return an ITT with headpart referencing the  results of BuildITT (p', Align)
                        and with children obtained from BuildITT(ci, null)
                              for each right sibling ci of the  real_head of p
   For ConcatCompHead:  Return an ITT obtained by invoking BuildITT(p, ConcatHead)
```

Figure 8. The BuildITT Function

Catenate takes two arguments: a ParseTree *p* whose initial sequence is to be concatenated, and a ParseTree *r*, beyond which the concatenation is to stop, based on the particular directive involved. It returns a pair <*Code*, *String*>. The *Code* indicates if the concatenation succeeded, failed, or is incomplete. If the concatenation succeeded, BuildITT creates an ITT with a headpart string containing the concatenated sequence, and children consisting of ITTs of the right-hand siblings of *r*.

Complex aspects of BuildITT and Catenate relate to coordinations within to-be-concatenated extents. The desired effect is to include short coordinations within the concatenation, while bracketing its boundaries and non-leaf components, e.g. " [{Old men} and women]", but aligning the elements of longer coordinations.

So if Catenate (in step 3) determines that the string resulting from a coordination is very long, it directly or indirectly returns an indicator to BuildITT that the concatenation failed. BuildITT (step 3d) then returns an ITT structured so that the sub-constituents that were to be concatenated are eventually shown as aligned, using different methods dependent on the directive *d*.

For example, if *d* is ConcatHead or ConcatSimpleComp, the result ITT contains:

a) a headpart reference to an ITT built by BuildITT(*p*',Align), where p' is like *p* but without the right-hand siblings of its *real_head*, and

b) children consisting of the ITTs for those right-hand siblings, if any.

5 TextTree Files and TextTreebanks

Previous sections focused on the production of individual TextTrees by the TextTree generator. This section considers some uses of the generator and auxiliary methods within parser development.

A particularly useful approach for producing parser output review material using the generator is sketched in Figure 10. In that approach, the best parses for a document are converted to standard ParseTrees expressed as XML entities and written to a file of such entities, interspersed with arbitrary other information. A separate, parser-independent process that includes the TextTree generator then creates a TextTree file by substituting TextTrees for the ParseTree entities.

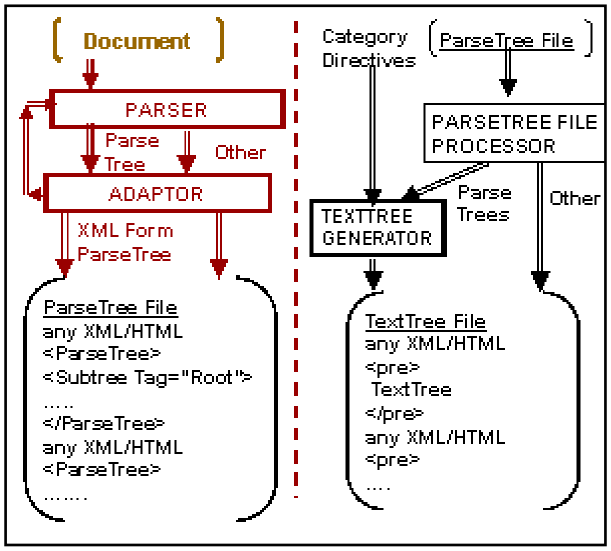


Figure 10. TextTree file creation

Such a process may be used to create the HTML TextTree file of Figure 1, which is a standard output form for the hybrid parser. The TextTrees are surrounded by HTML <pre> and </pre> tags to maintain the spacing and linebreaks produced by the generator. The interspersed information in this case consists of the sentence text and links to detailed displays of the best parse found, other parses with high preference scores, as well as the initial chunks.

Reviewing parse results for a document then consists of reading the TextTree file and, depending on circumstances, either simply noting or classifying the errors found for later debugging, or investigating them further via the links to the detailed displays.

5.1 TextTreebanks

Whatever the limitations (Carroll et al., 1998) of the various treebank-based bracket scoring measures derived from the Parseval approach of Black et al. (1991), they can be useful in monitoring parser development progress and in comparing the capabilities of different parsers, at least if there are large differences in scores.

But, as noted earlier, obtaining a fully labeled treebank for a specific domain or genre is generally a very labor-intensive process. A potentially less costly alternative is to create informal treebanks consisting of TextTree files corrected by manual editing.

Both corrected and uncorrected TextTree files can be converted to files consisting of fully-bracketed strings by a simple script that considers only the contained TextTrees. The script brackets the TextTrees so as to retain the explicit brackets, and to add brackets around each subtree, i.e., around each sequence of a line and the lines, if any, indented beneath it, directly or indirectly.

For example, a full bracketing of the TextTree of Figure 5 would be:

```
{However,}
{a simple transformation
     {of parse trees}
     {to text trees}}
{can expedite {these activities}}
```

The actual bracketed strings produced by the script are ones acceptable as input to the EVALB bracket scoring program (Sekine and Collins, 1997) with all brackets expressed as parentheses, and brackets added around words (apparently required but subsequently discarded by the program). Also, most punctuation is removed, to avoid spurious token differences. Then bracketed files deriving from noncorrected and corrected TextTree files can be submited to EVALB to obtain a bracket score.

Lest this approach be dismissed as overly sketchy, we note that the resulting brackets are similar to those resulting from a proposal by Ringger et al. (2004) for neutralizing differences between parser outputs and treebanks by bracketing maximal head projections, plus some additional mechanisms to further minimize brackets.

5.2 Preventing Bracketing Errors

To avoid bracketing errors resulting from imprecise spacing in manually edited trees, the TextTree indentations used are relatively wide. With indentations of five spaces, it is likely that an imprecisely positioned line will be placed closer to the desired level of indentation, so that an intelligent guess can be made as to the intent. For example, in:

```
Line a
     Line b
          Line c
  Line e??
```

the misplaced Line e begins at a point closer to the beginning of Line a than Line b. It is then reasonable to guess that Line e is sibling to Line a.

6 Experiments

This section describes two limited experiments to assess the efficiency of reviewing parser outputs for accuracy using TextTrees. One of the experiments also measures the efficiency of TextTreebank creation

6.1 First Experiment

The document used in the first experiment was the roughly 500-sentence "Executive Summary" of the 9/11 Commission Report (2004). After parsing by the hybrid parser, the expected TextTree file, excerpted in Figure 1, was created, reproducing each sentence and, for parsed sentences, the TextTree string for the best parse obtained, and a links to the detailed two-dimensional tree representation.

Of the 503 sentences, averaging 20 words in length, 93% received a parse. However, reviewing the TextTree file revealed that at least 191 of the 470 parsed sentences were not parsed correctly, indicating an actual parser accuracy for the document of at most 55%.

Reviewing the TextTrees required 92 minutes, giving a review rate of 6170 words per hour, including checking detailed parses for sentences where errors might have lain in the TextTree formatting.

That review rate can be compared to the results of Marcus et al. (1993) for post-annotation proofreading of relatively flat, indented, but fully labelled and bracketed trees. Those results indicated that:

> "... experienced annotators can proofread previously corrected material at very high speeds. A parsed subcorpus …was recently proofread at an average speed of approximately 4,000 words per annotator per hour. At this rate…, annotators are able to find and correct gross errors in parsing, but do not have time to check, for example, whether they agree with all prepositional phrase attachments."

While the two tasks are not exactly comparable, if we assume that little or no editing was required in proofreading, the ballpark improvement of 50% is encouraging.

6.2 Second Experiment

For the second experiment, we used the CB01 file[3] of the Brown Corpus (Francis and Kucera, 1964), and reviewed both the TextTree file and then, separately, the detailed 2D parse trees also produced.

While the parser reported that 91 of the 103 sentences, or 88%, received a parse, the review of the TextTree file determined that at most only 50 sentences, or 48.5%, received a fully correct parse. The review of the detailed parse trees revealed three additional errors.

The comparison of review times was less decisive in this experiment, with the rate for the TextTree review being 5733 words per hour, and that for the detailed 2D representation 4410 words per hour.

However, there were non-quantifiable differences in the reviews. One difference was that the TextTree review was a fairly relaxed exercise, while the review of the 2D representations was done with a conscious attempt at speed, and was quite stressful—not something one would like to repeat. Another difference was that scanning the TextTree file provided a far better cumulative sense of the kinds of problems presented by the document/genre, which might be further exploited by a more interactive format (e.g., using HTML forms) allowing users to classify erroneous parses by error type.

The experiment was then extended to check the extent to which TextTree files could efficiently edited for purposes of limited-function treebank creation.

For this purpose, to minimize typing when a sentence had no complete parse, the TextTree file included the list of chunks identified by the XIP parser. A similar strategy could be used with parsers that, when no complete parse is found, return an unrelated sequence of adjacent constituent parses. This is done by some statistical and finite-state-based parsers, as well as by parsers employing the "fitted parse" method of Jensen et al. (1983) or the "fragment parse" method described by Riezler et al. (2002).

[3] With part-of-speech tags removed

Editing the TextTrees for the 104 sentences, with an average sentence length of 21 words, required 83 minutes, giving a rate of 1518 words per hour. This might be compared with the average of 750 words per hour for the initial annotation of parses in the Penn Treebank experiment (Marcus et al., 1993) mentioned earlier.

After the TextTreebank was created, the bracketing script described in section 5.1 was applied both to the original TextTree file and to the TextTreebank, and the results were submitted to the EVALB program, which reported a bracketing recall of 71%, a bracketing precision of 84%, and an average crossing bracket count of 1.15.[4] Two sentences were not processed because of token mismatches. As expected, these scores were much higher than the percentage of sentences correctly parsed.

7 Related Work

Most natural language parsers include some provision for displaying their outputs, including parse tree representations, and/or other material, such as head-dependent relationships, feature structures, etc. These displays are generally intended for deep review of parse results, and thus attempt to maximize information content

Work on reducing review effort usually takes place in the context of developing treebanks by selection among, and/or manual correction of, parser outputs. In this area, the most relevant work may be the experiments of Marcus et al. (1993) using bracketed, indented trees. They found that annotator efficiency required eliminating many detailed brackets and category labels from the parser outputs presented. Other approaches rest, in whole or in part, on selecting among alternative two dimensional parse trees, such as the distinguishing-feature-augmented approach of Oepen et al. (2002), discussed in Section 2. As discussed in that section, however, two-dimensional tree displays are problematical for large trees, and it is not clear to what extent distinguishing feature information can expedite selecting among attachment choices.

[4] Sentences not receiving complete parses were submitted to EVALB without any brackets, contributing zero counts to the total # of correct constituents recalled.

Other related work deals with reducing measured differences between parser outputs and treebanks due solely to grammar style. As discussed in section 5, the bracketed material used in treebank-based measurement by Ringger et al. (2004) is similar to the bracketed material that would result from systematically bracketing TextTrees.

Finally, work by Walker (2001), intended not for parser/grammar development, but to facilitate reading and improve retention, produces text formats that bear some similarity to TextTrees, but are more closely attuned to spoken phrasing than syntactic form. The method uses complex segmentation and indentation strategies generally based on a combination of punctuation and closed-class words.

8 Directions for Further Development

We have described an implemented method for presenting parser outputs permitting fast visual inspection of the results of parsing long documents, as well as efficient editing to create informal treebanks. Although, because of their flattened, skeletal nature, TextTrees can hide some parser errors, we strongly believe, based on extended usage, that the convenience of reviewing TextTree files can contribute significantly to parser development efforts.

However, TextTrees are best suited to languages tending to a fixed word order and post-modification. To improve results for languages and language aspects that do not fall into this category, TextTrees might be augmented with highlighting and color to indicate syntactic functions. One way this could be done in a parser- and grammar-independent way is by adding a string-valued representation feature to the standard ParseTrees, and an additional set of directives to map the feature values to representation alternatives. For example, a subtree might be annotated with the feature Rep = "subject", and the additional directives might include <"subject", "blue">. Experimentation is needed to determine the usability of this approach.

Another topic to explore is the use of TextTrees in the creation of corpora annotated by deeper syntactic or semantic information. Because such information is generally expressed in forms that are even less readable than parse trees, a useful bootstrapping practice is to allow annotators to approve, or select among, parser output trees connected with the deeper information (King et al., 2003). TextTrees might be used to facilitate this process, with annotators either (a) interactively selecting among alternative TextTrees or, because there may be many alternatives, (b) editing a TextTree file containing at most one parse for each sentence (possibly chosen arbitrarily) and using the result for offline selection. Also, a parser used in the bootstrapping might refer to bracketed TextTreebanks to avoid pruning away elements of correct parses at intermediate points in parsing.

A third direction for further work is in extending the TextTree approach to deal with outputs of dependency-based parsers that do not produce constituent trees. While this should be a natural extension, an alternative system of features and directives would seem to be needed.

Acknowledgements

The author is indebted to Tracy Holloway King, John Sowa, and the anonymous reviewers for their very helpful comments and suggestions on earlier versions of this paper.

References

Salah Aït-Mokhtar, Jean-Pierre Chanod, and Claude Roux. 2002. Robustness beyond shallowness: incremental deep parsing, *Natural Language Engineering* 8:121-144 Cambridge University Press.

Ann Bies, Mark Ferguson, Karen Katz, and Robert MacIntyre. 1995. Bracketing Guidelines for the Treebank II Style Penn Treebank Project.

E. Black, S. Abney, D. Flickinger, C. Gdaniec, R. Grishman, P. Harrison, D. Hindle, R. Ingria, F. Jelinek, J. Klavans, M. Liberman, M. Marcus, S. Roukos, B Santorini, and T. Strzalkowski. 1991. A procedure for quantitatively comparing the syntactic coverage of english grammars. In *Proc 1991 DARPA Speech and Natural Language Workshop*. 306-311.

John Carroll, Ted Briscoe, and Antonio SanFillipo. 1998. Parser Evaluation: a Survey and a New Proposal. In Rubio et al., editors, *Proc First*

International Conference on Language Resources and Evaluation, Granada, Spain, 447-454.

W. Nelson Francis and Henry Kucera. 1964, 1971, 1979. Brown Corpus Manual. Available at http://helmer.aksis.uib.no/icame/brown/bcm.html Corpus available at http://nltk.sourceforge.net

Karen Jensen, George Heidorn, Lance A. Miller, Yael Ravin. 1983. Parse fitting and prose fixing: getting a hold on ill-formedness. *Computational Linguistics* 9(3-4):147-160.

Tracy H. King, Stefanie Dipper, Anette Frank, Jonas Kuhn, and John Maxwell. 2000. Ambiguity management in grammar writing. In Erhard Hinrichs, Detmar Meurers, and Shuly Wintner, editors, *Proc ESSLLI Workshop on Linguistic Theory and Grammar Implementation*, 5-19, Birmingham, UK.

Tracy H. King, Richard Crouch, Stefan Riezler, Mary Dalrymple, and Ronald M. Kaplan. 2003. The PARC 700 Dependency Bank, *Proc. 4th International Workshop on Linguistically Interpreted Corpora, at the 10th Conf of the European Chapter of the ACL*, Budapest.

Mitchell P. Marcus, Mary Ann Marcinkiewicz, and Beatrice Santorini. 1993. Building a Large Annotated Corpus of English: The Penn Treebank, *Computational Linguistics* 19(2):313-330.

Paula S. Newman. 2005. Abstract: TextTrees in Grammar Development. *Workshop on Grammar Engineering of the 2005 Scandinavian Conference on Linguistics (SCL).* Available at http://www.hf.ntnu.no/scl/grammar_engineering

9/11 Commission. 2004. Final Report of the National Commission on Terrorist Attacks upon the United States, Executive Summary. Available at http://www.gpoaccess.gov/911

Stephan Oepen, Dan Flickinger, Kristina Toutanova, and Christopher Manning. 2002. LinGO Redwoods: A Rich and Dynamic Treebank for HPSG, *Proc Workshop on Treebanks and Linguistic Theories (TLT02),* Sozopol, Bulgaria

Stephan Riezler, Tracy H. King, Ronald M. Kaplan, Richard Crouch, John T. Maxwell, and Mark Johnson. 2002. Parsing the Wall Street Journal using a Lexical-Functional Grammar and Discriminative Estimation Techniques. *Proc 40th Annual Meeting of the Assoc. for Computational Linguistics* (ACL), Philadelphia, 271-278.

Eric K. Ringger, Robert C. Moore, Lucy Vanderwende, Hisam Suzuki and Eugene Charniak. 2004. Using the Penn Treebank to Evaluate Non-Treebank Parsers, *Proc 2004 Language Resources and Evaluation Conference (LREC)*, Lisbon, Portugal.

Satoshi Sekine and Michael Collins. 1997. EvalB. Available at http://nlp.cs.nyu.edu/evalb

Randall C. Walker. 2001. Method and Apparatus for Displaying Text Based Upon Attributes Found Within the Text, U.S. Patent 6279017.