Towards a Language for Natural Language Treebank Transductions

Carlos A. Prolo
Department of Informatics and Applied Mathematics - DIMAp
Federal University of Rio Grande do Norte - UFRN
Natal, RN, 59078-970, Brazil
prolo@dimap.ufrn.br

Abstract

This paper describes a transduction language suitable for natural language treebank transformations and motivates its application to tasks that have been used and described in the literature. The language, which is the basis for a tree transduction tool allows for clean, precise and concise description of what has been very confusingly, ambiguously, and incompletely textually described in the literature also allowing easy non-hard-coded implementation. We also aim at getting feedback from the NLP community to eventually converge to a *de facto* standard for such transduction language.

1 Introduction

Linguists have always liked to use trees to theorize about the structure of natural language sentences (Marcus et al., 1983). In the 1990's, computer scientists also started to grow excited with the possibility of getting those proposed structures reasonably accurately from a computer. The advent of large corpora with annotated syntactic structures, the treebanks, among which the Penn Treebank (PTB) (Marcus et al., 1993; Marcus et al., 1994) is a notable representative, made it possible to build statistical parsers with an accuracy that was not feasible before. From the early work of (Magerman, 1994; Magerman, 1995; Charniak, 1997; Collins, 1997), state-of-the-art accuracy has been progressively raising, achieving now scores above 92% in the English Penn Treebank, as reported, for instance, in (Shindo et al., 2012).

As for dependency representations (Nivre et al., 2016; Nivre and Fang, 2017), although they capture syntactic structure in a different way, they are still generally represented as trees, even if not necessarily caring for the order among the children of a node.

Of course, not everybody is expected to agree on any given tree-structure style. On the linguistic side, each practitioner has their own theoretical conceptions on what is an adequate syntactic structure for a sentence, which may in turn differ from that of a specific treebank. Often all the information that is needed to promote the intended change is already present in the tree, either with some structure different than the desired, or "latent" in the annotation (Chiang and Bikel, 2002), in the form of secondary attributes on the nodes. When the change to the desired structure can be mechanically accomplished by applying to the trees a precisely defined rule, we call it a transduction. In order to perform more complex transformations we may instead define sequences of simpler transductions to be applied as a pipeline.

A few tree transducers have been proposed in the literature, among them the ones described in (Chiang and Bikel, 2002), based on context-free style rules, (Blahetta, 2003), based on Richard Pitto's **TGrep** (see (Rohde, 2005)), **Tsurgeon** (Levy and Andrew, 2006) and **TTT** (Purtee and Schubert, 2012). There are also transducers for other purposes, perhaps the currently most well known of them being W3C's **XPath** (World Wide Web Consortium, 2017), but those do not meet the needs of natural language and computational linguists practitioners (see for instance (Purtee and Schubert, 2012) for an initial discussion.) Although also built from Pitto-Rohde **TGrep** tools, the transduction language we propose in this paper is substantially diverse from that described in Blahetta's thesis (which, according to (Levy and Andrew, 2006) is no longer available).

This work is licensed under a Creative Commons Attribution 4.0 International License. License details: http://creativecommons.org/licenses/by/4.0/

Proceedings of the 27th International Conference on Computational Linguistics, pages 1022–1032
Santa Fe, New Mexico, USA, August 20-26, 2018.

Our main, initial motivation for designing the language and the tool we describe here came from perceiving the impact tree transduction can have in natural language parsing technology. There has been many really interesting proposals for parsers over these past 20 to 30 years, but what is even more striking is that they arose side by side with an intensive practice of what is often referred to as *hacking the trees*, which is now known to have played an important role in their success. For example, since his early work, Collins realized that it would be important, as a statistical parameter for deciding the correct tree structure of a sentence, to learn about the boundaries of *non-recursive* noun phrases, or *base NPs*, in the sentence.[1] However, his statistical parsing models would not be able to learn that particular parameter directly from the treebank, because base NPs were not explicitly marked as constituents in the tree, even if all necessary information was there: a human or a computer can easily follow some definition rules for base NPs, with no other external knowledge, and tell where they are in the annotated sentences. So, given a Penn Treebank tree like in Figure 1 the base non-recursive NPs, according to Collins, would be the following spans:

- (NP (DT A) (NNP SEC) (NN proposal) (S ...))
- (NP (NN reporting) (NNS requirements))
- (NP (DT some) (NN company) (NNS executives))
- (NP (DT the) (NN usefulness))
- (NP (NN information))
- (NP (NN insider) (NNS trades))
- (NP (DT a) (JJ stock-picking) (NN tool))

Collins realized that, inserting a node with a new label he called NPB to delimit the span, those constituents would help his training process so as to increase by a statistically significant amount his parser accuracy. After the transductions, the trees would be as following. Notice that whenever the entire NP span is itself immediately under some other NP, instead of adding a fresh NPB, the lower NP that dominates the spam is replaced with an NPB. That is the case in the first, third and the last two NPs below.

- (NP (NPB (DT A) (NNP SEC) (NN proposal) (S ...)))
- (NPB (NN reporting) (NNS requirements))
- (NP (NPB (DT some) (NN company) (NNS executives)))
- (NPB (DT the) (NN usefulness))
- (NPB (NN information))
- (NP (NPB (NN insider) (NNS trades)))
- (NP (NPB (DT a) (JJ stock-picking) (NN tool)))

The particular *tricks* used by Collins were reported in (Bikel, 2004). In fact, what once could have been perceived by some as a trick, is increasingly being acknowledged as an opportunity to recover relevant latent information, in principled ways, and making it available to the parser's training model.

Having this in mind we designed a general transduction language, based on Richard Pitto's **TGrep**.[2] We named it **tsed** after Unix **sed**, the string transduction tool counterpart to **grep**.

The obvious test suite was the set of Collins transformations as very precisely described (although in plain English, not in a formal language) in (Bikel, 2004), and also in Section 3.

Most tree transformations used in parsing have been hard-coded into the training and parsing algorithms. In particular, that was the case in the Collins implementation and even in Bikel's version. Although the latter claims to be highly parameterizable, which is indeed true to a great extent, one still has to hard-code in a programming language all the tree transformation processes.

In the following sections we describe the transduction language, and validate its expressive power replicating Collins's transformations as a test case. Of course the tool can be used for many other purposes as already mentioned. We conclude and discuss what is still planned for the future.

2 The Language

We describe here the transduction language. Appendix A contains a concise grammar for the language.

[1] See, for instance, (Collins, 2003) for his particular precise definition of the term.

[2] **TGrep** stands for *Tree Grep*, for the fact that it searches trees in a away which is parallel to the way Unix **grep** searches strings, and is currently available through Douglas Rohde's improved **TGrep2** version (Rohde, 2005).

```
(S (NP-SBJ (DT A) (NNP SEC) (NN proposal)
     (S
       (NP-SBJ (-NONE- *) )
       (VP (TO to)
           (VP (VB ease)
               (NP
                 (NP (NN reporting) (NNS requirements) )
                 (PP (IN for)
                     (NP (DT some)(NN company)
                         (NNS executives))))))))
   (VP (MD would)
       (VP (VB undermine)
           (NP
             (NP (DT the) (NN usefulness) )
             (PP (IN of)
                 (NP
                   (NP (NN information) )
                   (PP (IN on)
                       (NP (NN insider) (NNS trades) ))))
           (PP (IN as)
               (NP (DT a)(JJ stock-picking)(NN tool))))))
   (. .)  )
```

Figure 1: A Penn Treebank tree

2.1 Overview

A transduction is a pair (s, r) (or $s ==> r$ in the syntax of the language) where s is a *search expression* and r is a *replacement expression*. Together they define how to transform a treebank input tree into a new one.[3] The search expression describes what we look for in the input trees, and the replacement expression defines how to build the output tree based on the patterns found during the search phase.

The search expression is based on restrictions to be met by the input tree in order for a match to succeed. There are two kinds of patterns in the search expression: the *node specifications*, intended to match with nodes of the input tree that comply with the pattern; and the *operators* that relate the node patterns defining a layer of restrictions that has to be respected by the nodes. So, a search expression such as $NP < PP$ will look for a node with label "NP", that immediately dominates another node, labeled "PP". This syntax is strongly based on **Tgrep2**.

However, while **Tgrep** is designed only to find patterns, **tsed** has to handle transformations, so we added the concept of *placeholders*, in a somewhat similar way as Unix **grep** was extended into **sed**. When a transduction is applied to an input tree, if the search is successful we say that there is a *match*, and as a byproduct each placeholder will be assigned a subtree of the input. Of course, there can generally be several ways a match can succeed, leading to different assignments of placeholders into subtrees. For instance, in the tree of Figure 1 we can find three NPs that immediately dominate a PP and also the second of this NPs dominates the third one. Indeed it is a very important issue the precise semantics that govern the specifics of search order and disambiguation. This is the subject of a later section of this paper.

The *main placeholder*, marked by enclosing the node specifier with square brackets, also defines the point where the transformation will take place. So a transduction such as $[NP] < PP ==> dog$ will succeed in a tree that has an NP immediately dominating a PP. The match will assign to the main placeholder [] the subtree for the NP found. And the transformation phase will replace the whole subtree of the NP, by the single node "dog". We will see ahead how to express more complex replacement patterns. For now, we just want to note that if the pattern was $NP < [PP] ==> dog$, then the substitution would take place at the PP subtree and the NP would be preserved in the tree as well as all other trees sibling to the PP.

[3]The transducer tool assumes boot input and output trees in the Penn Treebank format - label bracketed, constituent trees representation format, - and this format is used in the examples throughout the paper, though, of course, this format is not a relevant conceptual issue.

2.2 Search Expression

The search expression is in fact a *primary node specification* and an optional Boolean layer of restrictions. This is very much imported from **TGrep2**, with minor changes. So, in the search expression $[NP](< NN| < NNP)\&\$.PP$, "|" means disjunction (logical "or"), "&" means conjunction (logical "and"), "<" stands for immediate domination, and "$." stands for the first node having as an immediate sibling the node to the right. Then the expression instantiates the placeholder [] to a subtree labeled NP that either immediately dominates (<) an NN or that immediately dominates an NNP, and, additionally, the NP has to have as an immediate sibling of its parent a node labeled PP.

The search expression allows embedded restrictions, so $[NP](< NN| < NNP)\&\$.(PP < (IN < of))$, forces that additionally to the restrictions above, the PP (not the NP) should immediately dominate a node labeled IN (the preposition in the English Treebank), and that preposition has to be "of". The reader familiar with **TGrep2** might notice that we used the parenthesis both to disambiguate precedence among the restrictions and for embedded restrictions.[4]

2.3 Place Holders

There are two types of placeholder: *cut* and *copy*. Cut placeholders are marked by enclosing the node specifier with square brackets in the search expression. Once they match and are assigned to a certain subtree, that subtree will be excised from its parent during the transformation, and will not be at its original place anymore in the output tree, unless explicitly reinserted according to the replacement expression. The copy placeholder uses curly brackets ($\{\}$) instead of square brackets. The subtree assigned to it can be copied elsewhere in the output tree, as specified by the replacement expression, but will not be removed from its original place. Copy and cut placeholders are numbered. The main or primary placeholder presented in the previous sections is assigned number 0 by default.

Currently **tsed** is centered in the replacement of the hole which is left in the original tree when the the subtree assigned to the main placeholder is excised. The other cut placeholders are not replaced. An example is given right below.

Advancing a little in the details of the replacement expression, the transduction $[NP] < [1 : PUNCT] \Rightarrow (NP\ [NP]\ [1 : PUNCT])$ helps us exemplify the concept of the cut placeholder. It assigns to [] (same as [0 :], the default). a subtree labeled NP that dominates a node PUNCT. This node PUNCT is assigned to the the other cut placeholder, with number 1 ([1 :]). The output tree will be formed replacing the whole NP tree (placeholder [] or [0 :]), by a new tree defined by the pattern in the replacement expression as a fresh NP, that has two children: the NP subtree matched to $[NP]$, and the subtree matched to $[1 : PUNCT]$. Moreover, the PUNCT subtree will be removed from below the old NP. The reader may have recognized this as a very simple mechanism for raising a node in the tree. If the expression were $[NP] < \{1 : PUNCT\} \Rightarrow (NP\ [NP]\ \{1 : PUNCT\}\}$, then the PUNCT subtree would not be excised from its original place and the effect would be to have two copies of it in the output tree.

At this point the reader should be wondering that the reference to the placeholders could be shorted, and indeed the above expressions are equivalent to $[NP] < [1 : PUNCT] \Rightarrow (NP\ []\ [1 :])$ and $[NP] < \{1 : PUNCT\} \Rightarrow (NP\ []\ \{1 :\})$. We call this shortened representations such as [], $\{1 :\}$ *back references*.

2.4 Replacement Expression

The replacement expression is a sequence of trees. Each tree has the same syntax as the PTB representation of a tree, except that we can use the placeholders instead of labels. In this representation, a leaf node is represented just by its label, such as "NN". A subtree which is not a leaf is represented, recursively by the sequence, enclosed in parenthesis, formed by the label of the subtree root followed by the representation of the subtrees of the root node from left to right. Thus a replacement expression such as

[4] In **TGrep2** embedded restrictions use square brackets instead. This change makes things more uniform, preserves the brackets for placeholders, and does not lead to ambiguity in interpretation. We also force explicit inclusion of & for conjunction which makes the search expression more readable, though, well, for the sake of compatibility with TGrep we allow dropping &.

$(PP\ (IN\ from)\ (NP\ (NNP\ Santa)(NNP\ Fe)))$ represents the usual English PTB representation for the prepositional phrase "from Santa Fe." Now, the reader can go back to the transductions of the previous subsection to recognize that this interpretation has been applied there, for instance to $(NP\ []\ [1\ :])$ as a subtree rooted at an NP, with two children subtrees. That is the main application for the placeholders and the back references.

The possibility of having a sequence of trees instead of just one, fits nicely the rest of the language. So if one wants to flatten a little more the English Penn Treebank, transductions such as $[NP] < (\{1 : NP\}\ \$.\ \{2 : PP\}) \Rightarrow \{1\ :\}\ \{2\ :\}$ replaces the NP subtrees that stand for an NP modified by a PP (the lower NP has an immediate right sibling PP) by a flattened one with the higher NP eliminated.[5] [6]

2.5 Restrictions

We have imported exactly the same very rich set of restriction operators defined in **TGrep2**. These are called "links"in the **Tgrep2** documentation. As for Boolean operators, we took the hard decision of only allowing negation as part of the restriction operators. So $! <$ means "do not immediately dominate, as in **TGrep2**. But using the negation higher in the search expression would be pretty confusing and non-intuitive to interpret in the transduction. [7]

2.6 Node Specification

Node specification in the transduction was one of the hardest issues in the design of **tsed**. Due the characteristics of treebank labels, we felt it would be desirable to have both pattern matching internal to the node label and some amount of editing capability.

In the examples so far we defined a node in the search expression as a constant string possibly enclosed in brackets or braces. This is very limited in many senses and we now extend the syntax in a few ways.

First, the part inside the brackets/braces is allowed to be edited in the replacement expression. Moreover fixed context patterns can be provided outside the brackets/braces, both to the left and to the right.

A node specification is then a triple (l, m, r) where m stands for middle and is the part enclosed in the brackets or braces; l is the left context and r is the right context. So the transduction. $[NP] - TMP \Rightarrow [NPT]$ looks for a node with label NP-TMP, and changes the NP portion of the label to [NPT], keeping the left and right context, therefore resulting in a label $NPT - TMP$. Notice that the left context in the example is empty, so the match will only occur for labels that start with the NP sequence of characters. Moreover, the matched label for the example can not have anything beyond the TMP suffix. In this sense, up to this point, label matching has to be exact given the pattern.

In each of the three parts a "*" matches with any character sequence and "?" is a wild card, like in file specifications in Linux. This is a pretty easy and powerful resource. Then $[NP] * -TMP* \Rightarrow [NPT]$ will replace all NPs that contain the feature TMP to NPT (maintaining the context). So an $NP - SBJ - TMP$ would be replaced to $NPT - SBJ - TMP$.

Escaped characters are allowed as well as substrings enclosed either in single or double quotes.

Finally, POSIX regular expressions are allowed protected by "//" as in **sed**. This provides a very great flexibility to node specification for the experienced user. We only should enforce as we said in the last paragraph, that that behavior is not like in **grep** or **TGrep2**, that looks for the pattern anywhere in the node. Instead it requires full match. So, a node specifier to match any node with the TMP feature should be expressed as something like $*/ - TMP/*$ or using only POSIX regular expression syntax $/. * -TMP. * /$, but not just $/ - TMP/$. The interesting point is that all the node-specifier expressions is ultimately converted to a POSIX regular expression, the semantics of which is widely known and very well documented. In the conversion, the separation of the left, middle and right part of the labels are provided by inserting parenthesis, supported by POSIX regex functions that then automatically locate them in the labels of the input tree.

[5] In fact a slight more complex expression should be used since this one is ignoring that extra material on the right of the PP or on the left of the lower NP would be lost.

[6] It is just an example, we are not advocating flattening the PTB!

[7] This was one of the few points where we found important to restrict the language in favor of usability and avoiding unpredictable behavior.

2.7 A Precise Semantics for the Transduction Driver

Avoiding non-deterministic behavior is a crucial issue in the project of artificial languages, often overlooked in its importance by people unfamiliar with language design and implementation. Indeed, it is an essential part of a language definition to provide clear rules specifying exactly what will be the output given any context of application of a valid transduction specification to a valid input tree.

For transduction languages based on patter matching it is essential to state clearly how the driver that executes the transformation works. For instance if we ask to eliminate a subtree labeled A that dominates a B, suppose that there happens to be in the corpus a tree with a path where an A immediately dominates a lower A, and the lower A immediately dominates a B. The question is whether the lower or higher subtree labeled A will be removed. Another issue is that of what to do after a match is performed: should it move on to the next tree, or should it try to reapply the transduction again to the same tree? And in the latter case, should it start it at the beginning of the output tree of the previous application? This issues could be treated as reasonable divergence among designers with regards to preference of behavior. However, it turns out that different behaviors are required depending on each particular situation. This section is intended to make precise the possible behaviors of the driver made available to the user by setting some execution options in the tool.

First, the search expression is seen as if parsed into an abstract, syntax tree like structure (see for example (Aho and Ullman, 1972)) in which operators are internal nodes and operands are their children. The leftmost leaf is the primary node specifier. Then the search is applied against input trees as if reading the syntax tree in an in-order traversal, that is, looking for a node, then looking for the restriction operator, and then to the subexpression to the right of the operator.

The Boolean layer is interpreted as in modern implementation of programming languages where the ”OR” is a ”conditional or” and the ”AND” is a ”conditional and”. This is essential to understand how the placeholders are assigned to subtrees. For instance once a full match succeeds on a tree, through a certain branch of an ”OR”, it will not try other paths. Then it will proceed according to the following algorithm:

1. Let *start* be the root of the input tree.

2. Assign *start* to *current node*

3. Search the input tree from the *current node* in a preorder left-to-right traversal order looking for a node that matches the primary node of the search expression (leftmost leaf of the syntax tree).

4. Proceed according to the syntax tree operators. Each operator dictates its one intuitive behavior. In a nutshell it searches from the *current node* ”outwards”in the tree. For example: ”dominates”search the next node in the same way as the primary node; ”is dominated”searches upwards till a match is found or it tries to move above the root; search for siblings move from the current node to the endpoints (left or right) In fact each operator has its own semantics which we cannot exhaust here, but the principles stated so far should be enough to understand most if not all of the behavior.

5. Whenever a restriction fails, backtrack. Failing branches in an OR layer remove internal assignments made to placeholders on this branch.

6. If the match fails, output the tree as it is at that moment.

7. If the match succeeds,

   (a) Apply the replacement expression, modifying the current tree.

   (b) Assign to *current node* the node that would follow, in a preorder left-to-right traversal, the one that replaced the primary placeholder after the transformation just concluded. Usually this is the leftmost child of the root of the subtree inserted there, unless these subtree is just a leaf.

   (c) Resume at step 3

The way we defined at 7.(b) the reentrant behavior to resume the process after a successful match transformation seems to be the most desirable from our modeling experience so far. It is a compromise to an alternative behavior of restarting over again from the root, which causes many non-termination problems and unpredictable behavior.

However two alternatives still being considered as options at the command line are: to resume from the leftmost child of the node matching the primary node specifier (which usually is the primary placeholder, though not always); and also resuming from the node which would be the next, in a preorder left-to-right traversal, after the whole subtree substituted at primary placeholder was visited. This latter behavior is extremely safe in that it would completely prevent non-termination. And it is certainly faster. But it generally does not correspond to the user's desired behavior.

2.8 Back references

Back references are essential at the replacement expression, but they also have an important use in the search expression. Because we require back references to be previously declared and have strict rules of scope of visibility enforcing the well-formedness of the search expression we do not experience the strange behaviors reported in (Rohde, 2005) by the expression "crossing link".

2.9 Ranges of subtrees and endmarkers

There are situations where being able to express tree-range fillers may be very useful. Tree-range fillers are represented by "..." enclosed as placeholders. Thus, the transduction $[NP] < (\{1 : NP\}(\$, \{3 : ...\}\&\$.(\{2 : PP\}\$.\{4 : ...\}))) \Rightarrow \{3 :\}\{1 :\}\{2 :\}\{4 :\}$ is a more robust version of the flattening example above. $\{3 :\}$ matches with the sequence of siblings to the left of the lower NP, and $\{4 :\}$ matches with the sequence of siblings to the right of the PP, thus the extra material below the higher NP is preserved. Of course there are other ways of doing this. However this greatly simplifies the transductions that have to eliminate intermediate nodes.

Another feature is the endmarker which stands for the lack of nodes. So $\# < SBAR$ means an SBAR at the root of the tree. Similarly the endmarker can be used to capture the notions of leaf node and right- and leftmost child.

2.10 "Semantic"consistency of the Boolean layer with respect to placeholders

Every artificial language has those issues related to allowing only well-formed expressions. For instance in **tsed**, we enforce that a placeholder cannot be defined twice in the search expression, and that a back reference can only appear if the corresponding full placeholder has been defined earlier in the expression. We are not going to exhaust these aspects here. However it is important to notice that the definition and use of placeholders interact very dangerously with the logical operators. For instance, a back reference in a restriction in a branch of a logical OR cannot refer to a placeholder defined in an earlier branch. This is not a problem if the restrictions are in the branches of an AND operator. Natural language semanticists will quickly perceive this asymmetry which is generally not an issue in programming language implementation. Another issue is that a placeholder is visible outside of an OR layer of restrictions only if it has been declared in all branches. We have considered very intuitive and indeed interesting rules of scope and visibility for the placeholders but will refrain from entering in a morass of detail that would be required to explain them here.

3 A Case Study

We present in this section examples of transductions that implement Collins pre-processing steps as described in in (Bikel, 2004). All the quotations in this section are from his article, with references to figures removed to avoid confusion.

3.1 Base NP node insertion

This transformation was exemplified earlier in the previous section. Bikel describes it as follows:

An NP is basal when it does not itself dominate an NP; such NP nodes are relabeled NPB. More accurately, an NP is basal when it dominates no other NPs except possessive NPs, where a possessive NP is an NP that dominates POS, the preterminal possessive marker for the Penn Treebank. These possessive NPs are almost always themselves base NPs and are therefore (almost always) relabeled NPB. For consistency's sake, when an NP has been relabeled as NPB, a normal NP node is often inserted as a parent nonterminal. This insertion ensures that NPB nodes are always dominated by NP nodes. The conditions for inserting this "extra" NP level are slightly more detailed than is described in Collins' thesis, however. The extra NP level is added if one of the following conditions holds:

- *The parent of the NPB is not an NP.*
- *The parent of the NPB is an NP but constitutes a coordinated phrase.*
- *The parent of the NPB is an NP but*
  - *the parent's head-child is not the NPB, and*
  - *the parent has not already been relabeled as an NPB.*[8]

The transformation is accomplished by the following **tsed** transduction sequence. We well apply the sequence to the following PTB annotated sentence:

```
(S (NP-SBJ (NP (NP (DT The) (NNP SEC))
               (POS 's))
           (NNP Mr.) (NNP Lane))
   (VP (ADVP-MNR (RB vehemently))
       (VF disputed)
       (NP (DT those) (NN estimates)))
   (. .))
```

1. We first replace NP labels that dominate some POS with a new fresh label POSNP: We find some NP nodes which dominates (operator <<, not necessarily immediate domination) a POS node as stated in the search string. The $*$ means that the labels, either POS or NP, may be followed by any character sequence, so it can match, say, an NP-SBJ. The brackets [] are used as placeholders once a match is found. That means that (1) the subtree rooted at the NP node is the one going to be replaced; (2) the brackets have been matched to that subtree in case we want to refer to that tree in the replacement string; and (3) we have the chance to replace the part of the string inside the brackets in case we want to use it. In this case it selects only the NP part of the label, not the additional suffix. That is because the "*"is outside the brackets. The replacement string tells us then that the same subtree will be kept there (because it puts the brackets back there). But the NP part of the label is replaced with POSNP, so, if that was an NP-SBJ, it will become POSNP-SBJ.

   search string: '[NP]* <<POS*' **replacement string:** '[POSNP]'
   output:

```
(S ( POSNP-SBJ ( POSNP (NP (DT The) (NNP SEC))
                       (POS 's))
               (NNP Mr.) (NNP Lane))
   (VP (ADVP-MNR (RB vehemently))
       (VF disputed)
       (NP (DT those) (NN estimates)))
   (. .))
```

   The search is made in a preorder traversal and finds all matching occurrences, each one being subject to the replacement.

2. In this rule we replace with NPB the labels of NPs that do not dominate another NP (those are the base NPs). The exclamation mark means negation as in **Tgrep**. Notice that we have already preclude the possibility of targeting NPs dominating POS, since we changed their names to POSNP.

   search string: '[NP]* !<<NP*' **replacement string:** '[NPB]'
   output:

```
(S (POSNP-SBJ (POSNP ( NPB (DT The) (NNP SEC))
                       (POS 's))
               (NNP Mr.) (NNP Lane))
   (VP (ADVP-MNR (RB vehemently))
       (VF disputed)
       ( NPB (DT those) (NN estimates)))
   (. .))
```

[8] Only applicable if relabeling of NPs is performed using a preorder traversal.

3. Then we switch the POSNP labels back to NP.

search string: '[POSNP]*' **replacement string** '[NP]'

output:

```
(S ( NP-SBJ ( NP (NPB (DT The) (NNP SEC))
                 (POS 's))
             (NNP Mr.) (NNP Lane))
   (VP (ADVP-MNR (RB vehemently))
       (VF disputed)
       (NPB (DT those) (NN estimates)))
   (. .))
```

4. And finally we add an extra NP node on top of the NPB, in case it is not immediately dominated by some other NP (again following **TGrep**, a single > means that it has to be **immediately** dominated in order to match.

search string '[NPB]* !>NP*' **replacement string** 'NP <[NPB]'

output:

```
(S (NP-SBJ (NP (NPB (DT The) (NNP SEC))
                 (POS 's))
             (NNP Mr.) (NNP Lane))
   (VP (ADVP-MNR (RB vehemently))
       (VF disputed)
       ( NP (NPB (DT those) (NN estimates))))
   (. .))
```

The reverse transformation is described by Bikel as:

> *I*n post-processing, when an NPB is an only child of an NP node, the extra NP level is removed by merging the two nodes into a single NP node, and all remaining NPB nodes are relabeled NP.

This is easily accomplished by a single transduction rule

search string '[NPB]* !>NP*' **replacement string** 'NP <[NPB]'

We have implemented also other transductions from (Bikel, 2004) and the reverse versions whenever well defined. Some of them depend on location the head node on a constituent. That could be accomplished through a pre-processing tool that could mark the head nodes with a "-HEAD"tag. We have considered replicating the head finding rules but we have not done it yet.

4 Conclusion and Future Work

Of course the appeal of tree transductions is not only its use for parsing pre- and post-processing tasks. We used this application in this paper to show that **tsed** language is sufficiently powerful so that it can be used to replace transformations originally made by programming language code pieces, that were never subject to a restriction in power of expression. That is, Collins defined and coded the transformations that he thought as useful and that he could mechanically express having all the Turing machine power of a general programming language at his disposition. Those transformations can be elegantly and succinctly captured by a special purpose transducer language, stripping them out of the main learning algorithm.

By inspection, we extensively observed that the application of all the transductions generated the new trees as we think they should. In fact there is no easy way to directly check against what Collins would generate, because the latter are never explicitly realized. They are internal to the parsing process. We have considered instead, comparing the parsing accuracy of the original parser with that of the parser with the transformations made by **tsed**. Unfortunately this requires removing the corresponding pieces of code in either Collins' or Bikel's implementation, and we have not done it yet.

The examples in the test case have been successfully implemented in an existing implementation that have some limitations in expressive power. Still it could model the rules described in (Bikel, 2004).

Tsed will be used in projects to experiment on parsing accuracy enhancement through sound corpora transformations.

We hope to get feedback from the NLP community joining efforts to eventually converge to a *de facto* standard for a transduction language.

References

Alfred V. Aho and Jeffrey D. Ullman. 1972. The Theory of Parsing, Translation, and Compiling, volume I: Parsing. Prentice-Hall, Englewood Cliffs, NJ, USA.

Daniel Bikel. 2004. Intricacies of Collins' parsing model. Computational Linguistics, 30:479–511.

Don Blahetta. 2003. Function Tagging. Ph.D. thesis, Brown University.

Eugene Charniak. 1997. Statistical parsing with a context-free grammar and word statistics. In Proceedings of the Fourteenth National Conference on Artificial Intelligence, pages 598–603, Menlo Park, CA, USA.

David Chiang and Daniel M. Bikel. 2002. Recovering latent information in treebanks. In Proceedings of the 19th International Conference on Computational Linguistics (COLING'2002), pages 183–189, Taipei, Taiwan.

Michael Collins. 1997. Three generative, lexicalised models for statistical parsing. In Proceedings of the 35th Annual Meeting of the Association for Computational Linguistics, pages 16–23, Madrid, Spain.

Michael Collins. 2003. Head-driven statiscal models for natural language processing. Computational Linguistics, 29(4):589–637.

Roger Levy and Galen Andrew. 2006. Tregex and tsurgeon: tools for querying and manipulating tree data structures. In Proceedings of the Fifth International Conference on Language Resources and Evaluation (LREC), Genoa, Italy.

David M. Magerman. 1994. Natural Language Parsing as Statistical Pattern Recognition. Ph.D. thesis, Stanford University.

David M. Magerman. 1995. Statistical decision-tree models for parsing. In Proceedings of the 33rd Annual Meeting of the Association for Computational Linguistics, pages 276–283, Cambridge, MA, USA.

Mitchell P. Marcus, Donald Hindle, and Margaret M. Flack. 1983. D-theory: Talking about talking about trees. In Proceedings of the 21st Annual Meeting of the Association for Computational Linguistics, pages 129–136, Cambridge, Massachusetts, USA, June. Association for Computational Linguistics.

Mitchell P. Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. 1993. Building a large annotated corpus of English: The Penn Treebank. Computational Linguistics, 19(2):313–330.

Mitchell Marcus, Grace Kim, Mary Ann Marcinkiewicz, Robert MacIntyre, Ann Bies, Mark Ferguson, Karen Katz, and Britta Schasberger. 1994. The Penn Treebank: Annotating predicate argument structure. In Proceedings of the 1994 Human Language Technology Workshop.

Joakim Nivre and Chiao-Ting Fang. 2017. Universal dependency evaluation. In Proceedings of the NoDaLiDa Workshop on Universal Dependencies, UDW@NoDaLiDa 2017, Gothenburg, Sweden, May 22, 2017, pages 86–95.

Joakim Nivre, Marie-Catherine de Marneffe, Filip Ginter, Yoav Goldberg, Jan Hajic, Christopher D. Manning, Ryan T. McDonald, Slav Petrov, Sampo Pyysalo, Natalia Silveira, Reut Tsarfaty, and Daniel Zeman. 2016. Universal dependencies v1: A multilingual treebank collection. In Proceedings of the Tenth International Conference on Language Resources and Evaluation LREC 2016, Portorož, Slovenia, May 23-28, 2016.

Adam Purtee and Lenhart Schubert. 2012. Ttt: A tree transduction language for syntactic and semantic processing. In Proceedings of the EACL 2012 Workshop on Tree Automata Techniques in Natural Language Processing, Avignon, France.

Douglas L. T. Rohde, 2005. TGrep2 User Manual.

Hiroyuki Shindo, Yusuke Miyao, Akinori Fujino, and Masaaki Nagata. 2012. Bayesian symbol-refined tree substitution grammars for syntactic parsing. In Proceedings of the 50th Annual Meeting of the Association for Computational Linguistics, pages 440–448, Jeju, Korea.

World Wide Web Consortium. 2017. Xml path language (xpath) 3.1. `http://www.w3.org/TR/xpath-31/`, March.

Appendix A. A reference grammar for tsed language

A concise Yacc-style grammar can be very helpful as a reference to the language even without any semantic restriction or explanation (a grammar is worth a thousand words!) Also shown are the main lexical items in Lex-style.

```
start:  search_expression  turnsto  replacement_expression

search_expression:  search_primary   opt_restrictions  ;
opt_restrictions: restrictions |  ;
restrictions: restrictions terminal_or restrictions_and
            | restrictions_and   ;
restrictions_and: restrictions_and terminal_and restrictions_not
        | restrictions_and restrictions_not /* same as AND */
        | restrictions_not ;
restrictions_not: terminal_not restrictions_not /* "not" is under scrutiny */
        | lpar restrictions rpar
        | restriction ;  /* break precedence, [] on tgrep*/
restriction:    operator search_second  ;
search_second: lpar search_expression rpar  |  search_primary ;
search_primary: node_specifier | node_range_filler ;
node_specifier: nodename | end_marker ;

replacement_expression: tree_seq;
tree_seq: tree_seq tree  |  tree   ;
tree: node |  // folha
      LPAR node tree_seq RPAR ;
node: new_node |                        /* similar to nodename */
      primary_placeholder_instance |
      auxiliary_placeholder_instance |      ;

/* nodename specification in lex */
  {NODEBLOCK}
  {NODEBLOCK}?"["[[:blank:]]*"]"{NODEBLOCK}?
  {NODEBLOCK}?"{"({NUMBER}\:)?[[:blank:]]*"}"{NODEBLOCK}?
  {NODEBLOCK}?"["({NUMBER}\:)?[[:blank:]]*"]"{NODEBLOCK}?
  {NODEBLOCK}?"["{NODEBLOCK}"]"{NODEBLOCK}?
  {NODEBLOCK}?"{"({NUMBER}\:)?{NODEBLOCK}"}"{NODEBLOCK}?
  {NODEBLOCK}?"{"({NUMBER}\:)?[[:blank:]]*\.\.\.[[:blank:]]*"}"{NODEBLOCK}?
  {NODEBLOCK}?"["({NUMBER}\:)?[[:blank:]]*\.\.\.[[:blank:]]*"]"{NODEBLOCK}?

/* Helpers */
ALFANUM                 [[:alnum:]-]
WILDCARD                \?
ANYSEQUENCE             \*
ESCAPED                 \\.
DQSTRING                \"([^"\\]|\\.)*\"
SQSTRING                \'([^'\\]|\\.)*\'
REGEX                   \/([^/\\]|\\+[^\\])+\/
NODEBLOCK          ({ALFANUM}|{WILDCARD}|{ANYSEQUENCE}|{ESCAPED}|{DQSTRING}|{SQS
```