An Implementation of the TAG Formalism for Parsing With Construction Grammar

Roshan Pulapura

August 2020

1 Introduction

In my previous paper, *An Implementation of Filler-Gap Constructions Using the Tree-Adjoining Grammar Formalism*, I explored the theory of Construction Grammar as a viable alternative to the standard Chomskyan principles of Universal Grammar (UG) as a way of syntactically analyzing sentences. Although Construction Grammar has been thought to perform well on the types of sentences on which UG-based syntactic theories tend to fail, Construction Grammar is relatively under-studied in the field of Linguistics and comparatively little research has been done to formalize and evaluate Construction Grammar as a basis for syntactic parsing. Therefore, I proposed the Tree-Adjoining Grammar (TAG) formalism as a natural pairing for Construction Grammar and, through a case-study in deriving filler-gap sentences, showed that (1) the principles of Construction Grammar could be captured using a TAG analysis of a sentence and (2) that the filler-gap constructions, as implemented in TAG, were able to avoid the issues of syntactic overgeneration and semantic generalization that arise in UG-based wh-movement analyses of these same sentences. Although new issues of under-generation and unwieldy specificity arose from

the Construction Grammar approach, I ultimately concluded that it still merited further study due to its significantly increased accuracy on notoriously difficult to parse sentences.

The goal of this research is to build upon a couple of the clearest limitations of the previous paper. Firstly, the decision to formalize Construction Grammar with TAG was not comprehensively motivated from a logical perspective and was instead justified retroactively based on its effectiveness in the case study. This project takes a more rigorous approach to describing the precise tree-deriving operations which can be done to generate a grammatical tree in the TAG associated with a given natural language, and further clarifies the relationship between the elements of the natural language TAG (i.e. the terminal and nonterminal symbols, sets of initial and auxiliary trees, the substitution and adjunction operations, the function that determines which nonterminal symbols can be substituted into other nonterminal symbols) and the analogous fundamental principles of Construction Grammar (defining grammatical constructions and speech utterances, and compositional, inheritance, and recursive relationships between constructions). Since the previous paper demonstrated the effectiveness of TAG in producing a Construction Grammar analysis through worked out examples in the filler-gap case study, this follow-up paper justifies this pairing of Linguistic theory and logical formalism from a more logical and mathematical perspective.

The other primary limitation of the previous study was that it was entirely theoretical in nature, and potential practical applications of the research, whether in the field of Computational Linguistics or otherwise in engineering/NLP, were left for future studies. This paper attempts to evaluate the practical applications of Construction Grammar by constructing a algorithm that parses filler-gap sentences using TAG, as theoretically described in the previous paper. This algorithm defines a TAG tree type and implements the adjunction and substitution operations. Ultimately, it is able to take a series of elementary

TAG trees from various constructions and find a way to combine them via TAG operations into one large, correct TAG tree. Then, after defining a mapping between words in a limited vocabulary and all the different elementary trees which those words may anchor, applying the TAG algorithm to the elementary trees corresponding to words of an input sentence yields, as hypothesized, a correct Construction Grammar parse of the input sentence. This leads me to conclude that (1) the TAG formalism (and algorithmic implementations of its operations) is effective for producing CxG parses both theoretically and practically, and (2) for Linguistic research on the syntactic-semantic interface, an area in which Construction Grammar is believed to be well suited, a parser primarily concerned with accuracy even in niche cases such as this one could be used to preprocess the corpus (just as UG-based parsers are used currently). Subsequently, statistical analysis may be done on which positions/contexts certain words or constructions are commonly used in and so forth. However, the necessity of hard-coding likely millions of individual elementary trees to completely define a given natural language and the time complexity of the parsing algorithm make this framework unlikely to be successful for engineering/NLP applications, such as document understanding and summarizing, in comparison to statistical or machine learning methods, which are more concise to write and more efficient to run on large amounts of text.

2 Background

This paper assumes a degree of familiarity with the details of the TAG formalism for complete understanding. However, Chapter 3, which provides the definition of the TAG for a natural language and the inference rules for generating TAG trees, will provide a description of independent and auxiliary TAG trees and substitution and adjunction operations.

This background section will briefly restate the motivation and principles behind Con-

struction Grammar and will reproduce the definitions of the filler-gap constructions developed in the previous paper. It will then highlight some prior research on TAG parsers and other implementations of Construction Grammar to situate this study, which develops a TAG parser to implement Construction Grammar.

2.1 Construction Grammar

Construction Grammar is a theory of Grammar developed by George Lakoff, Charles Fillmore, and Paul Kay at the University of California, Berkeley in the 1980s. The essence of this theory is that syntactic and semantic elements of language are acquired by children in the same way that lexical items/vocabulary words are: that is, there is some learned relationship between a form (a speech utterance) and a function (the meaning indicated by the speech utterance), whether that form is a word or a syntactic structure (Goldberg 2013: 3-4). This theory is in opposition to the Chomskyan theory of Universal Grammar, which proposes some innate human capacity for language and that all grammatical sentences within any language can be produced following whichever subset of the innate syntactic rules applies to the speaker's native language. Then, acquisition of syntax involves learning which subset of those innate syntactic rules belong to the speaker's native language.

There are four basic principles of Construction Grammar (Hoffman 2013: 2-4):

1. Firstly, a grammatical construction is defined as a form-function pair between a morphological, lexical, syntactic, or semantic element and its meaning. Therefore, lexical constructions would simply be the pairing between the word and its meaning. However, a syntactic construction can be thought of as a relationship between syntactic trees with related meaning. For example, the verb phrase (VP) construction is a form-function pair between syntactic structures that contain a subject and a verb, and the semantic interpretation of the subject carrying out the action desig-

nated by the verb (regardless of what the actual subject and verb are lexically, this meaning of 'subject performing verb' is inherent to the syntactic structure).

2. Secondly, constructions do not form derivational relationships with one another. In other words, we never define a construction by decomposing another individual construction and reorganizing it with a different shape or order. For example, we would NOT say that the wh-question 'Who did she see?' is derived from a move operation on the simple sentence 'She saw [DP].' Instead, constructions form compositional relationships (where one construction is formed by putting together other constructions in a specific order) and inheritance relationships (where every member of a subtype construction is also a member of the supertype construction and can be used in any semantic context requiring the supertype construction).

3. There are no 'deep structures' corresponding to surface structure speech utterances. That is, there are no 'move' operations in construction grammar, and syntax trees are formed by combining other trees, not by moving elements around in a given tree.

4. All constructions (both lexical and syntactic/semantic) are language-specific. Just like each natural language has a unique set of vocabulary words, but several languages may have similar/related words to indicate the same meaning, each natural language also has a unique set of syntactic structures, but several languages may likewise have similar/related syntactic structures to indicate the same meaning.

2.2 Filler Gap Constructions

In my previous paper, I developed grammatical constructions to represent several different types of filler-gap sentences because filler-gap sentences highlight some of the weaknesses of UG in comparison to CxG. That is, the UG approach generalizes the same Wh-

move operation for different types of filler-gap sentences which have vastly different semantic meaning (the implication of contrast in topicalized clauses, the emphasis in a Wh-exclamative, the interrogative sense in Wh- direct and indirect questions, etc). Since these structures largely employ the same lexical items (who, whom, what, which etc) to instantiate, clearly this divergence in meaning comes from the syntax, but Wh-movement generalizes the same syntactic operations for vastly different types of semantic structures (Sag 2010:489-495) .

Secondly, the simple move operation rules, without any further restrictions or exceptions, can generate all grammatical filler-gaps– but they also generate additional ungrammatical filler-gaps. The CxG approach, on the other hand, ensures that only correct sentences will be accepted.

In my previous paper, I defined four types of filler-gap constructions, namely topicalized clauses, Wh-exclamatives, Wh-interrogatives, and Wh-relatives. Each of these filler-gap constructions is composed of a filler, which is a nonverbal complement or adjunct, and a main clause, which is a verbal and contains a GAP element in either the complement or adjunct of the verb (matching with the filler in its complement/adjunct nature).

Topicalized clauses contain AdjP, AdvP, DP, or PP in the filler and a linear sentence main clause (with the GAP in the complement for DP and in the adjunct for AdjP, AdvP, PP). Wh-exclamatory clauses contain what-a-DPs, what-pl-DPs, How-AdjPs, and How-AdvPs in the filler and linear sentences in the main clause (with GAP in the complement for the DPs and in the adjunct for the Adj/AdvPs). Wh-interrogatives can be either direct questions or indirect questions: both types of interrogative accept the same types in the filler: Who-NP, Whose-DP, What-DP, What-NP, Which-DP, Whom-NP, What-Pl-DP, How-AdjP, How-AdvP, Wh-PP, When-PP, Where-PP, Why-PP. The difference is that the direct questions require an inverted sentence as the main clause, while the indirect questions

require a linear sentence or an infinitive. As expected, the AdjP, AdvP, and PPs require a GAP in the adjunct, while the NPs and DPs require it in the complement. Finally, Wh-relatives are themselves auxiliary trees rather than initial trees and may be adjoined to DP. They contain Who-NP, Whom-NP, Which-NP, Whose-DP, That-NP, Wh-PP in the filler and a linear sentence in the main clause. It can also have Wh-PP in the filler and an infinitive in the main clause. As with the other types of filler-gap, the GAP occurs in the adjunct of the verb for the PPs and in the complement for the NPs and DPs. Here are some examples of each of the different types of filler-gaps to situate the reader:

(2.1) Top. Clause: The girl, I saw (but the boy, I did not).

(2.2) Top. Clause: At the store, Sarah bought grapes (but online, she bought clothes).

(2.3) Wh Exclam.: How quickly Robin runs!

(2.4) Wh Exclam.: What a good book I read.

(2.5) Wh Direct Q: What book did I read?

(2.6) Wh Indirect Q: I wonder what book she read.

(2.7) Wh rel: Isabel wrote the book which Allison read.

(2.8) Wh rel: I went to the store at which books are sold.

2.3 Prior TAG Parsers (XTAG Group)

Before producing my own TAG-based parser, I would like to contextualize this research within the larger subject area of TAG parsing. The most notable TAG parsing algorithms have been developed by the XTAG group at the University of Pennsylvania (where the TAG formalism was also initially developed). XTAG is a lexicalized Tree-Adjoining Grammar for the full English language (Champollion 2007: 1). XTAG itself handles filler-gaps by trying to produce syntax trees that resemble those produced by Wh-movement. The XTAG approach avoids many of the associated island effects, however, by placing

substitution restrictions in the filler clause. Nevertheless, the XTAG group lays out some instances in which their parser overgenerates. One particular example that stands out is sentences such as 'the way how to solve the problem' (Bleam 2001: 150). This particular type of overgeneration is easily addressed via Construction Grammar. There are no elementary trees in the relative clause construction which have a How-AdvP as the filler. Meanwhile, since XTAG does not consider How-AdvP to be a distinct 'construction' and instead treats it as part of a broader category of question words which accept move operations, it is harder to restrict sentences such as 'the way how to solve a problem.' In terms of the parser itself, XTAG uses an adapted version of the CYK algorithm used to parse the related formalism, Context Free Grammars. This algorithm is a bottom-up dynamic programming based algorithm. It essentially generates all the possible trees anchored by each individual word in the sentence and then generates all the trees that can be formed by combining a sequential pair of words, and then all the trees that can be formed by combining those sequential pairs, and so on (Alonso 1999: 151-153). This algorithm is very similar to the one I have used in terms of combining elementary trees. The primary difference between the XTAG parser and mine is that I have a CxG based set of elementary trees which are a lot more specific than the ones used by XTAG.

2.4 Prior Construction Grammar Implementations (FCG)

The TAG formalism is not the only way Linguists have attempted to formalize Construction Grammar. Luc Steels, the developer of the Fluid Construction Grammar formalism (which his research group designed specifically to represent Construction Grammar) also has an associated parser for FCG. The Fluid Construction Grammar parser is a constraint based algorithm, and a correct parse for a sentence is determined by unifying the constraints of each component construction. Essentially, for each word, FGC determines which constructions it is statistically likely to belong to. Then, the FCG parser determines which larger construction or combination of constructions, which can be formed by com-

bining the component constructions in the given order, is most statistically likely. For example, if 'ate' is likely to be a component of a transitive VP construction or an intransitive VP construction, and 'the cake' is likely to be a subject of a transitive or intransitive construction or the direct object of a transitive construction, combining these two words together would result in the transitive construction with 'ate' as the verb and 'the cake' as the direct object because this is the only combination where both components belong to their likely constructions in the proper positions. There may be incidences as well where multiple compositions are equally likely, so the parser keeps track of that and ultimately chooses the parse for the entire sentence which is most likely (Steels 2017: 178-182).

3 The TAG Formalism and Construction Grammar

3.1 Introduction

This chapter will demonstrate how to define any natural language grammar using the Tree-Adjoining Grammar formalism, adapted to include links between nodes, lexicalization, and substitution constraints. Given a set of elementary initial and auxiliary trees for a natural language, the following definitions will show how to grammatically combine the language's elementary trees to produce more complex grammatical utterances. Furthermore, it will define the notion of a 'grammatical construction' and a 'Construction Grammar,' as introduced in the literature review, in terms of sets of initial or auxiliary trees within the TAG and, furthermore, formalize the recursive, compositional, and inheritance relationships between these constructions.

3.2 Defining a Language with the TAG Formalism

A natural language grammar is defined as a lexicalized, linked, and constrained Tree-Adjoining Grammar $G = (\Sigma, V, S, I, A, f, L)$, where:

Σ is the finite set of all terminal symbols. A terminal symbol indicates that no further adjunctions or substitutions may be made at that node. Within a natural language grammar, the terminal symbols are lexical items (such as words, abbreviations, acronyms that function as words, etc.) as well as a language specific set of valid null symbols such as the null pronoun PRO or the empty object GAP in conventional English linguistics.

V is the finite set of all nonterminal symbols. All nodes in a valid TAG tree, aside from leaf nodes, must be labeled with a nonterminal symbol. Leaf nodes may also be labeled with nonterminal symbols (in trees that represent incomplete sentences). All substitution and adjunction operations occur at nodes labeled with nonterminal symbols. In a natural language, nonterminal symbols represent types of phrases or clauses, including basic phrases such as the noun phrase (NP) and the verb phrase (VP), as well as more complex structures such as Wh-questions (Wh-Q) and Wh-exclamatives (Wh-Ex) which will be analyzed in the following chapter.

S is a designated start symbol. A complete sentence must have this symbol, or a valid substitution for this symbol, labeling the head node. This symbol need not be the head of every valid tree in G– note that the majority of elementary trees as well as valid composite trees represent incomplete sentences. However, a parse of a grammatical sentence will always begin with the start symbol.

I is the set of initial trees as defined for LTAG. The head symbol need not be S, but all trees in I must have one leaf node (called the foot) labeled with a terminal symbol (called the anchor). All other leaf nodes may be labeled with either terminal or nonterminal symbols.

A is the set of auxiliary trees as defined for LTAG. The head node must be labeled with the same nonterminal symbol as one of the leaf nodes (the foot). A different leaf node must also be labeled with a terminal symbol (the anchor). All other leaf nodes may be

labeled with terminal or nonterminal symbols.

$f : (V \times V) \rightarrow \{0, 1\}$ is a function which takes in two nonterminal symbols and returns 1 if the second input may be substituted for the first and 0 otherwise. For example, in English grammar, nouns may be used in place of noun phrases, so $f(NP, N) = 1$. However, determiner phrases may not be used in place of noun phrases, so $f(NP, DP) = 0$.

L is a set of triples (t, m, n) where $t \in I \cup A$ and $m, n \in \Sigma \cup V$. L represents the links drawn between nodes m, n in a specific tree t. As noted in the previous paper, m must c-command n (that is, n is the descendant of the sibling of m). Furthermore, if n is labeled with a nonterminal symbol, if it has any children, they must be labeled with terminal symbols.

Adjunction and substitution operations in a natural language TAG are mainly unchanged from a standard TAG. As with a standard TAG, an auxiliary tree with head and foot labeled with nonterminal symbol v may be adjoined into any other tree at a node also labeled with v. Likewise, any tree with head labeled with nonterminal symbol v may be substituted into any other tree at a leaf node also labeled v following the same rules as standard TAG.

However, the natural language TAG also allows for the substitution of a tree with head labeled w into another tree's leaf node labeled with v, provided that $f(v, w) = 1$. For example, a tree with an NP leaf may accept a substitution of N or NP into that position, but a tree with an NP leaf may not accept a substitution of DP into that leaf. Therefore, we can say that the substitution of a tree with head node labeled w may be substituted into a leaf labeled with nonterminal symbol v iff $f(v, w) = 1$. We can expect that $f(v, v) = 1$ for a natural language since a node with a given nonterminal label should always be accepted at another node with the identical nonterminal label.

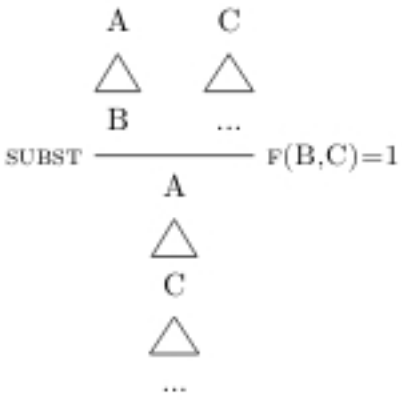
This property is slightly more complex for adjunction. Note that an auxiliary tree with head and foot labeled N may not be adjoined into a node labeled NP, since that would require both the substitution of a N for an NP (the N head of the auxiliary tree into the position of the excised NP) AND the substitution of an NP into an N (re-inserting the excised NP into the N foot). While the former substitution is acceptable, the latter is not. Therefore, an adjunction of a tree with head and foot labeled w at a node labeled v is valid iff $f(v, w) = 1$ and $f(w, v) = 1$.

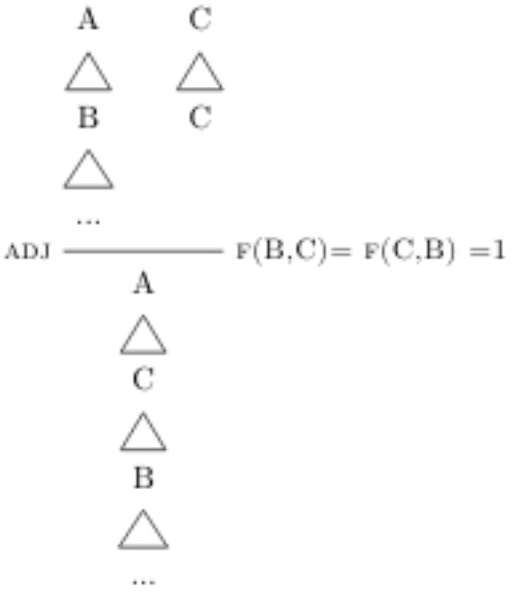
Since a natural language grammar also contains links, it must be noted that the direct links are preserved after application of the substitution or adjunction operations. If a node labeled v is either element of a link and receives a substitution, the head of the substituted tree replaces v in the link for the combined tree. If a node labeled v is either element of a link and receives an adjunction, the head of the adjoined tree replaces v in the link for the combined tree– v does not keep the link when it is re-inserted into the foot of the adjoined tree.

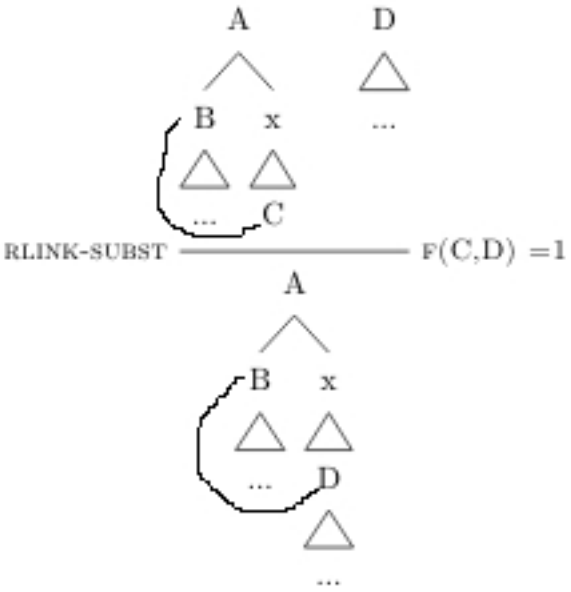
Furthermore, note that all links are initially defined in G and are associated with some elementary tree (an element of either I or A). This means that linking is not an operation and that all links are inherent to fundamental structures of language. This decision is justified by the Construction Grammar philosophy of surface-structure exclusive syntax that eliminates derivational relationships between constructions. For example, if we were to compare the basic subject-verb-object sentence 'Derek knows him (no link between subject and object) with the reflexive 'Derek knows himself' (pronoun-antecedent link between subject and object), Construction Grammar posits that these two structures belong to distinct constructions (although both inherit from the transitive verb sentence construction). Therefore, more complex reflexives could be produced from substituting/adjoining to the reflexive initial tree, while more complex SVO sentences could be produced from the SVO initial tree. However, construction grammar theorizes that reflexive sentences

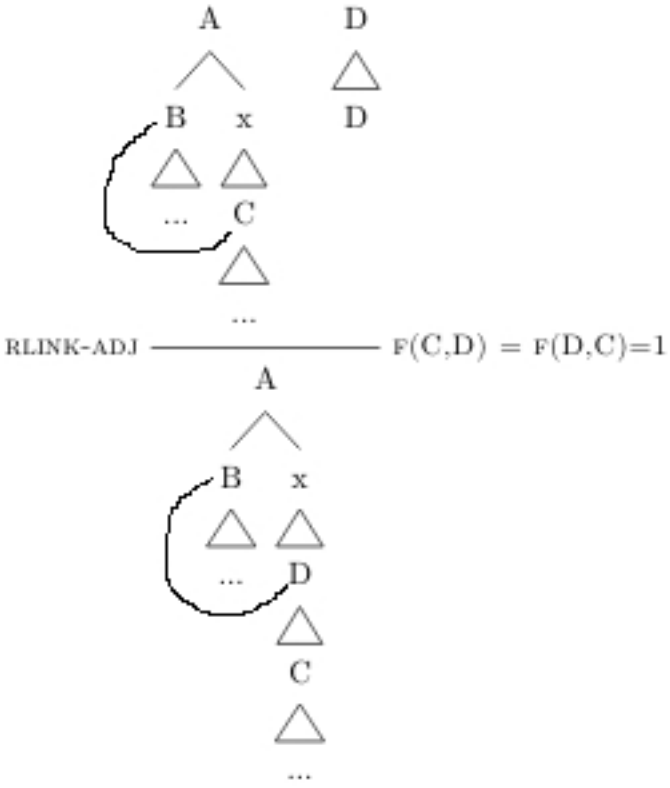
are NOT produced by mentally modifying a deep-structure SVO during interpretation.

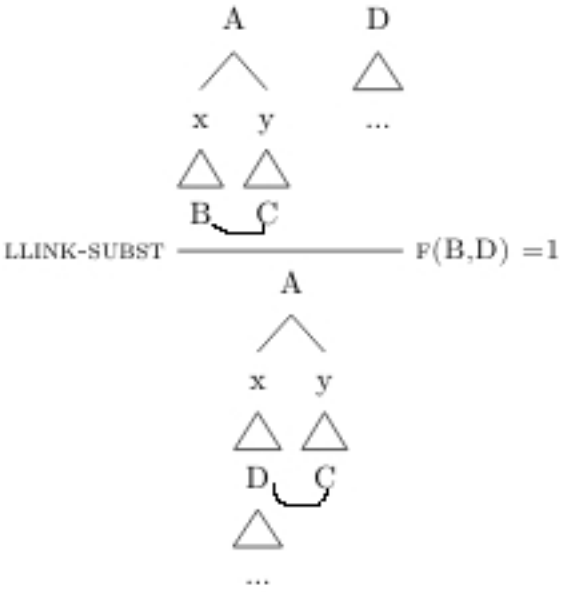
The above descriptions of the TAG operations can be concisely demonstrated using the following six inference rules:

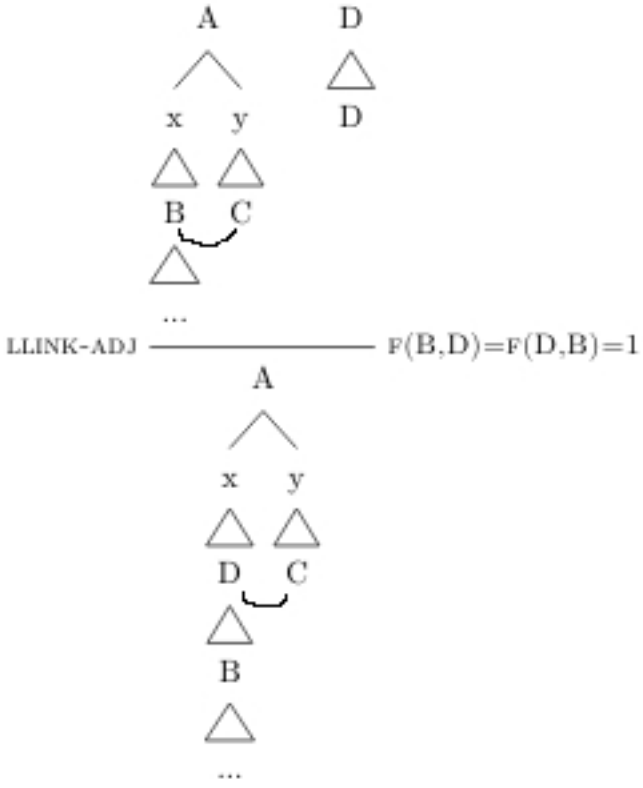





A
D
B
x
...
C
RLINK-SUBST
F(C,D) =1
A
B
x
...
D
...


A
D
B
x
D
...
C
...
RLINK-ADJ
F(C,D) = F(D,C)=1
A
B
x
...
D
C
...


A
D
x
y
...
B
C
LLINK-SUBST
F(B,D) =1
A
x
y
D
C
...


A
D
x
y
D
B
C
...
LLINK-ADJ
F(B,D)=F(D,B)=1
A
x
y
D
C
B
...

Finally, we will define the notion of grammaticality for a language represented by a TAG G. A tree represents a grammatical sentence if it is both *complete* and *correct*. A tree t is complete iff all leaf nodes are labeled with terminal symbols, whether they are words or language-specific null lexemes, and its head is labeled with either S or some $v \in V$ such that $f(S, v) = 1$. A tree t is correct if it can be derived from substitution or adjunction of elementary trees, as demonstrated by the inference rules.

Note that a correct tree need not be complete: 'The NP went home' is the correct substitution of the DP 'the NP' into the transitive VP sentence 'DP went home,' but the NP child of the DP is empty. Likewise, a complete tree need not be correct: 'The went went home' has terminal symbols in all leaf nodes, but a verb(went) was incorrectly substituted into a noun phrase in 'The NP went home' while $f(NP, V) = 0$ in English (and in most other languages).

3.3 A TAG Approach to Construction Grammar

Now that the notion of a grammatical sentence has been defined and rules have been provided for combining grammatical utterances correctly, this next section will demonstrate how the essential features of the Construction Grammar philosophy are represented in the natural language TAG described above.

Definition 1. Speech Utterance. *A Speech Utterance l can be represented as a list $[n_0 \dots n_n]$ of words, where the words are indexed in the order that they are spoken.*

The above definition simply serves to represent speech concisely for the purposes of defining grammaticality in a Construction Grammar. The corresponding speech utterance for a given syntax tree is simply the list of terminal symbols from left to right in that syntax tree.

Definition 2. Grammaticality. *A speech utterance l is a grammatical sentence under the natu-*

ral language TAG G if and only if there exists a `complete` *tree formed by* `correct` *substitution and adjunction of elementary trees in G, whose terminal nodes in order from left to right are equivalent to that speech utterance.*

This definition follows naturally from the definition of grammaticality presented in the previous section– spoken sentences are grammatical if they represent a tree with a grammatical derivation. Note that this definition goes in both directions– all grammatical sentences are generated from complete and correct trees, and no ungrammatical sentences are generated from complete and correct trees.

Definition 3. Grammatical Construction. *A Construction within a TAG grammar G is a nonempty subset of either the initial trees I or the auxiliary trees A in G.*

The essential component of a Construction Grammar, the construction, is simply a set of initial or auxiliary trees which carry some degree of semantic similarity, regardless of which lexical items are substituted into those trees.For example, the tree for a transitive verb (1), and intransitive verb (2), and a ditransitive verb (3) all belong to the VP construction and share semantic meaning with slightly different syntactic structure. Note that the above definition is slightly incomplete as it stands, since it does not account for adjoining auxiliaries into elementary trees. This will be addressed in subsection 2.4.

(3.1) The boy **slept**.

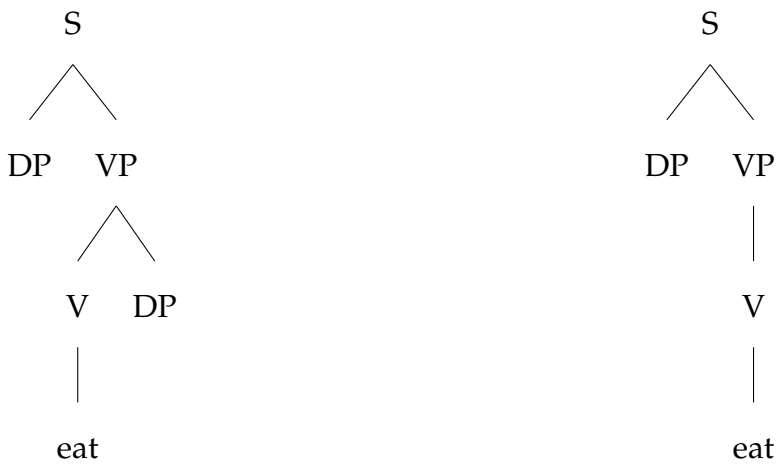
(3.2) The boy **ate the cookie**.

(3.3) The boy **put the cookie in the jar**.

In other words, the Construction Grammar philosophy is simply a way of grouping together trees which are semantically related but not syntactically derived from one another. All the verb phrases above are initial trees and cannot be transformed into the others via

adjunction or substitution (the intransitive V tree has one child, the transitive has two, the ditransitive has three), but they are nevertheless obviously related.

Note that trees anchored by the same lexical element may belong to different constructions, and most trees belong to several constructions. For example, the following two initial trees anchored by the word 'eat' may be drawn. The first belongs to the intransitive verb construction. The second belongs to the transitive verb construction. Both trees, however, belong to the verb phrase construction.



As a matter of convention, initial trees will be rooted in the name of the construction to which they belong. Auxiliary trees, however, must be rooted in the type they modify so that they may be adjoined above those types. For example, note that the tree for adverb phrases is rooted in VP, since adverbs are adjoined into VP to modify the verb.

Definition 4 (Inheritance)**.** *A construction C_1 is said to inherit from construction C_0 if $C_1 \subseteq C_0$.*

Theorem 1 (Substitution and Inheritance)**.** *If construction C_1 inherits from construction C_0, any tree belonging to C_1 may be correctly substituted into any position in which a member of C_0 is required. That is to say, $f(C_{0,}\, C_{1)} = 1$.*

The above definitions show the analogy between constructions within Construction Grammar and nonterminal symbols in TAG. If a leaf node is labeled with a given nonterminal symbol, any tree whose head is that nonterminal symbol may be substituted into that position. That is to say, the nonterminal symbol labeling the leaf node specifies a set of initial trees (i.e. those which may be substituted into that position), essentially designating a construction.

Therefore, the inheritance relationship between the original construction and the subset of

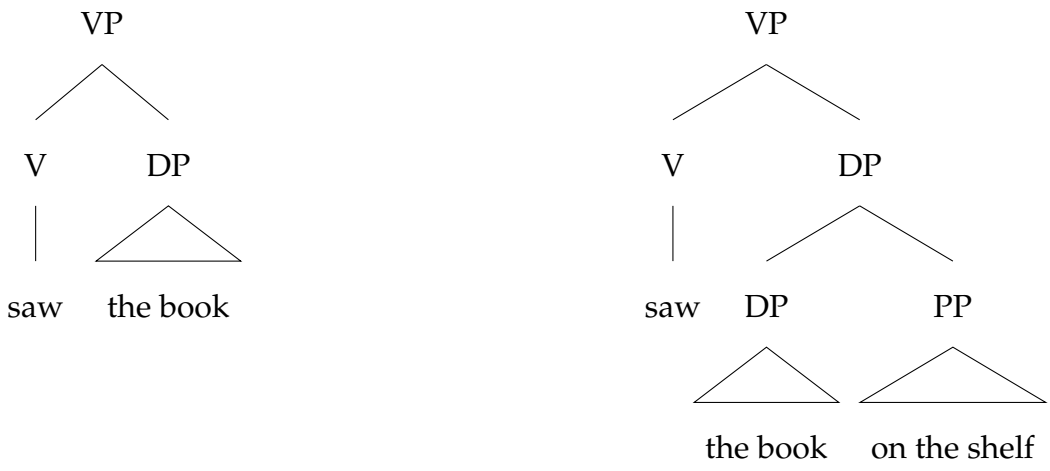
that construction just means that every element of the subset also belongs to the original set, so all of those elements may be substituted where a member of the original set is required. Thus supertype C_0 and subtype C_1 constructions are labeled with nonterminal symbols such that the substitution of a C_1 head into C_0 is permissible.

Definition 5 (Composition)**.** *Given an instance c of construction C and the ordered list of constructions $C_s = [C_0 \dots C_n]$, C_s composes c if, for all immediate children (in order) $d_0 \dots d_n$ of the root of c, d_i is an element of C_i.*

Linguistically speaking, if we speak a DP (for example, 'the dog') and then a VP (for example, 'barked') in succession, we are producing an S ('the dog barked'). This represents the concept of composition. When members of specific constructions are combined together in a given order, a member of a new construction will be formed. For example, the list $[Adj, NP]$ composes an instance of $AdjP$, since an adjective phrase is produced by combining an adjective and a noun phrase. Note that not every element of a construction is composed by the same list of other constructions. If we return to the example of transitive, intransitive, and ditransitive verbs, we see that VP can be produced from $[V]$, $[V, DP]$, and $[V, DP, DP]$.

3.4 Recursion

As described in the introduction section, language is not a finite set of speech utterances. There are infinitely many grammatical sentences which can be produced via recursion. For example, theoretically an infinitely long grammatical sentence may be produced from a sequence of prepositional phrases ('I saw the the book on the shelf in the store on the highway in New Jersey' etc). It is easy to see how the recursive nature of language is captured by the TAG formalism– the adjuction operation can theoretically be applied infinitely to a given initial tree by successively adjoining more and more auxiliaries to the same (or different) nonterminal nodes.



We will note then, that the second tree with the adjunct included is still headed by the nonterminal symbol VP and can therefore be substituted into any position calling for VP. It would then be logical that the second tree should belong to the VP construction. As we previously defined it, constructions are sets of elementary trees. Therefore, we offer a small modification:

Definition 6 (Construction (Updated))**.** *Let c be a subset of either the initial trees I or the auxiliary trees A in a natural language TAG, G. A tree t_0 belongs to a construction C corresponding to c if $t_0 \in c$ OR t can be formed by adjunction of an auxiliary tree $a \in A$ to some $t \in c$.*

What this definition means is that a construction is a set of elementary trees AND all the trees which can be formed via adjunction into those elementary trees. That is to say, 'the dog' and 'the big dog' are both members of the determiner phrase construction, and 'walked' and 'walked quickly' are both members of the verb phrase construction.

Now that TAG has been adapted for natural language within the framework of the Construction Grammar theory, the following sections will address developing a functional parser based on this theoretical work.

4 A TAG Parser for Filler-Gap Constructions

4.1 Introduction

The goal of these next two chapters is to demonstrate the effectiveness of the TAG-CxG approach I developed in my previous research in correctly parsing difficult sentences. Similarly to my prior work, I have decided to focus specifically on filler-gap sentences in this research because they provide an interesting opportunity to examine unconventional theories where standard theories have consistently failed. As mentioned in prior chapters, one of the major failings of the most popular Wh-movement based models in addressing filler-gap sentences is that these models tend to overgenerate sentences which are not grammatical. By simply following Wh-movement rules, so-called Wh-islands and other ungrammatical sentences will be produced, meaning that an extensive set of exceptions to the initially defined Wh-movement rules become necessary to ensure correct parses. Therefore, using the case study of filler-gap sentences, I demonstrated how the Construction Grammar approach could be used to correctly analyze sentences on which Wh-movement fails.

To take that one step farther, I have now developed an algorithm that parses filler-gap sentences based on Construction Grammar theory, using the implementation in the TAG formalism that I developed. Firstly, I defined a TAG tree type and created functions to perform the TAG operations on instances of the TAG tree type. Next, I wrote a backtracking algorithm that takes in a list of trees and outputs the tree that is formed from combining all the trees in the list in the given order via TAG operations, if such a tree is possible. I created a mapping between lexical items and the elementary trees that they anchor and defined several function to instantiate elementary trees anchored by given lexical items. Finally, I used the TAG operations to generate a parse tree for a given list of lexical items, i.e. a sentence.

This chapter will provide a detailed explanation for how each of the components of this algorithm works and will motivate some of the design choices. The following chapter will discuss how effective this TAG-based parser was in capturing the essence of Construction Grammar, this parser's ability to characterize difficult Wh-island effects and other instances of overgeneration, and how this methodology compares to conventional linguistic methodologies.

Two small caveats about this implementation: firstly, since this meant to be an extension of the filler-gap case study, I have not included further constructions for parsing other aspects of grammar (beyond the very basic structures like NP, VP, S, etc). This study also deals primarily with syntax and therefore is not meant to handle morphological details such as subject-verb agreement (i.e. it will not reject 'the children walks' which is treated the same as 'the children walk'). Finally, although links are crucial to the natural language TAG I defined in the previous chapter, since this parser deals exclusively with filler-gap sentences (and the links occur on the same nodes, neither of which can accept substitutions or adjunctions, across the board for all FG constructions, as demonstrated in the previous paper), they don't really offer much additional meaning in this case.

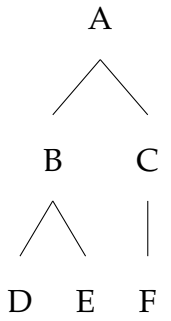
4.2 The TAG Tree Type and Helper Methods

This section will address the TAG `tree` type and the various methods used to access/modify elements of this type. The `tree` type has two cases, namely `Leaf` which contains only a node of type string, and `Tree` which contains a node of type string, and a list of other trees. In the `Tree` case, the string represents the nonterminal symbol placed at that node, and the list contains all the children of that node (which are also `tree` types).

The `get_head` method returns a node containing the nonterminal symbol stored in the head of the input tree. The `add_child` method inserts the second input as the child of the first input. Note that this is not a TAG operation but just a helper method to make

writing out initial trees easier– it simply appends the second input to the list of children of the first input.

The `to_string` and `opt_to_string` methods convert a `tree` and a `tree option` respectively to strings. The string representation is such that the head of a given tree is printed followed by a list of all of its children in brackets. For example, [A[B[D E] C[F]]] would be the string representation of the following tree:



There are also several other string/printing methods– `print_slist` which prints a string representation of a string list, `print_tree_list` which prints the `to_string` representation of each tree in a list, and `print_parses` which prints the `to_string` representation of each tree in a list of tree lists. All of these methods work by the same general mechanism of pattern matching on a list, printing the head, and then calling the same method on the tail.

There are a few other helper methods: `fold_tree` works essentially like `List.fold_left` but for trees. It traverses a tree and accumulates all the nodes following a given function f. This method was used to implement `count`, which traverses a tree and increments a counter for each node to count the total number of nodes in a tree, and `concat`, which traverses a tree and appends each node's label into a string to produce a string representation of the preorder traversal of the tree. Both of these methods were primarily used for debugging purposes.

4.3 TAG Operations: Adjunction and Substitution

Both the `adjoin` and `subst` methods (for adjunction and substitution) take in two trees, a parent and a child, where the child is to be substituted or adjoined into the parent. Both of these methods return option types– if the operation is successful, `Some Tree` is returned, and otherwise, `None` is returned. This allows the parser later on to try combining trees and proceed based on whether the operation succeeds.

The `adjoin` and `subst` methods are both dependent on the `f_compare` method. The `f_compare` method serves as the function f that is part of the definition of a TAG. This is the function that takes in two nonterminal symbols (i.e. nodes in this implementation) and determines if the second may be substituted for the first. Within `f_compare` a node may always be substituted into another node with the same label. All other substitutions I have hardcoded to reflect the substitutions that are acceptable within the CxG approach to filler-gap sentences.

The `adjoin` and `subst` methods are both dependent on the `fold_option` method as well. This method takes in a function from `tree` to `tree_option` (i.e. like `adjoin` and `subst` applied to a `tree`), as well as an input list and an accumulator. It attempts to apply the input function to each tree in the input list. If it succeeds (i.e. if f outputs `Some Tree` rather than `None`), it returns the same list as the input list, except with the individual element to which f was successfully applied changed (returned as a `tree list option` rather than as a `tree list`). If f applied to every element yields `None`, then `None` is returned.

The logic behind this method is that it allows the `adjoin` and `subst` methods to insert the child exactly once into the parent tree. For the `subst` method, if the parent tree is a leaf, if the child can be substituted into that leaf, the resultant tree (option type) is returned, and otherwise `None` is returned. If the parent tree is not a leaf, however, the child

tree must be substituted into one of the parent's existing children. Using `fold_option` it finds the first existing child of the parent into which the new child can be substituted, performs the substitution, and returns a `tree list option` of the new list of children with the substitution performed on one child and all the rest unchanged.

`adjoin` works in much the same way, except rather than simply substituting into leaves, the child tree can be adjoined at any point. So if the parent is a leaf, `adjoin` proceeds as `subst` does. If the parent is not a leaf, if the child and the parent can be substituted into one another (i.e. they can be adjoined) the adjoining is completed by inserting the parent back into the foot of the child. If the head of the parent is not compatible with the child for adjoining, this method proceeds as `subst` did, calling `fold_option` to find the first existing child of the parent into which the new child can be adjoined, and if it exists, performing that adjoin and leaving the remaining children unchanged.

These three methods provided the basic framework to build and combine trees using the TAG operations. The next step, then, was to develop an algorithm to determine if a whole series of trees might be combined in succession to form one large tree, and if so, to perform that combination.

4.4 Backtracking Algorithm to Combine List

When given a list of TAG trees, the `join_list`, `permutations` and `join` methods are used to combine those trees into one tree via the TAG operations, if such a tree is possible. These methods also all depend on an input sentence (`string list`), which corresponds to the initial trees they are attempting to combine. That is, each of the trees in the list is anchored by one of the words in the sentence. A proper combination of all the trees in the list, then, should not only be valid under the TAG rules, it should also produce a tree whose leaves correspond to the words in the sentence. Further detail on the initial trees anchored by lexical items is provided later in this section.

The `join` method takes in two trees and performs all four possible TAG operations to combine them (first substituted into the second, second substituted into the first, first adjoined into the second, second adjoined into the first). For each successful operation (i.e. returns `Some Tree` and not `None`) of the four, it then verifies the new tree against the input sentence, determining if the terminal nodes in the combined tree occur in the proper order. Further explanation of the verification method occurs in the next subsection. Ultimately, `join` outputs a `tree list` containing all the valid combinations of the input trees with the terminal symbols (i.e. the words) occurring in the correct order with respect to the input sentence.

The `permutations` method takes in two lists of trees and outputs every way of combining some tree from the first list with some tree from the second list. This method is necessary because there may be multiple trees corresponding to a given speech utterance (particularly incomplete sentences). For example, just the word 'eat' may correspond to a transitive verb phrase or an an intransitive verb phrase. Without seeing the next word, there is no way to know whether 'eat' is meant to be transitive (i.e. 'I will eat the cookie' or intransitive (i.e. 'I will eat'). Therefore, when parsing, both the transitive parse and the intransitive parse should be considered possibilities. Therefore, the permutations method combines all the possibilities for one word/series of words with all the possibilities for the next word/series of words, to get a `tree list` representing all the possibilities for those words/series of words together. For example, combining 'The boy' (which has just the DP possibility) and 'ate' (which should have the transitive and intransitive possibilities), should yield two trees for 'the boy ate' (the transitive and intransitive possibilities for 'ate' are both still possible). Note that `permutations` also takes in the `string list` representing the sentence as a parameter and verifies that all combined trees have lexical items in the correct order with respect to the sentence.

Finally, the `join_list` method takes in the full list of `tree lists`, where each `tree`

`list` represents the possibilities for a given word in the sentence. It returns a list of all possible trees that can be formed by joining all the input `tree lists` together via TAG operations. For example, for the sentence 'the boy ran', `join_list` would take a list consisting of a list of all possibilities for 'the', a list of all the possibilities for 'boy' and a list of all the possibilities for 'ran', and return a list of all possibilities for 'the boy ran', making sure to verify that all tree combinations end up with the lexical items in the same order as the input sentence. The `join_list` method is essentially a backtracking algorithm: for the base cases, if the initial `tree list list` is empty, it returns an empty list. If there is only one list in the `tree list list`, it returns that one list. If there are two lists in the `tree list list`, it returns the possible combinations of those two elements (i.e. by calling permutations). For the recursive case, it first attempts to combine the first two elements into a phrase, and recurses on a new list where the first element is that combined phrase (and the rest is the same). In the 'the boy ran' example, it would take the list for 'the', the list for 'boy', and the list for 'ran', get the list for 'the boy', and then call itself on the `tree list list` containing the lists for 'the boy' and 'ran'. If that recursive call yields at least one parse for the whole sentence, that parse is returned. Otherwise, it recursively calls itself on the original `tree list list` with the exception of the first element. If that yields any successful parses, they are combined with the first element via `permutations`. Essentially what this algorithm is doing is determining for each word whether it can be combined with the word before it to ultimately yield a parse, or whether it can be combined with the word after it to ultimately yield a parse, or neither.

4.5 Verifying the Parse with the Original Sentence

As mentioned in the previous subsection, the final parse needs to be a correct combination of trees for each word in the original sentence IN THE SAME ORDER as they were originally presented. That is, if the leaf nodes are taken from left to right in the final parse, they should form the original sentence. To ensure this result, the `remove_gaps`,

read_tree, and verify methods are used.

The remove_gaps method, as expected, simply traverses a string list and removes all occurrences of 'GAP'. This is necessary because a correct parse of filler-gap sentences will have at least one instance of the null lexeme 'GAP' while the original spoken sentence of course will not. Therefore, when comparing the list of terminal/lexical items from the parse to the original sentence, the 'GAP' instances are removed from the parse.

The read_tree method extracts all the lexical/terminal node labels from the input tree. It takes a simple pattern match approach on the input tree: if it is a leaf, if the label is a lexical item (i.e. not in the list 'phrases' of nonterminal symbols) it is appended to the output list, otherwise, if it is a tree, the method is recursively called on each child using fold_left.

Finally, the verify method determines if the first input string list is a sub-list of the second. Essentially, the algorithm determines if the second input list is a sub-list of the first list beginning at the first index of the first list, then the second index of the first list, and so on. For example, verify would return true on inputs ['the'; 'boy'; 'ran'] and ['boy'; 'ran'] but false on ['the'; 'boy'; 'ran'] and ['ran'; 'boy'].

The join method mentioned earlier makes use of these three methods to ensure that a combined tree is only accepted if the lexical items occur in the correct order for the sentence. For each of the possible combinations, it uses read_tree to extract the lexical items, calls remove_gaps to remove any 'GAP' instances which would diverge from the original sentence, and then calls verify on the resultant list of lexemes and the original sentence to determine if the list of lexemes is a sub-list of the original sentence, returning true if so and false if not. If verify returns true, that combination of trees is accepted.

4.6 Identifying Complete Sentences

Since this parser is intended to find the correct parse (or in rare cases, parses) of a given grammatical sentence, the `get_leaves`, `is_complete`, and `remove_incompletes` functions are used to identify which parses output from `join_list` represent complete sentences.

`get_leaves` as the name suggests takes in a tree and outputs a `string list` containing the labels of all the leaf nodes in the input tree. Some of these labels may be nonterminal symbols– if so, the input tree is incomplete. Since we are parsing complete sentences, the correct parse should not have empty categories (i.e. if our input is 'the boy ate', the output parse should have the intransitive VP with no DP complement, rather than the transitive VP with an empty DP complement, since in the given sentence, 'ate' is used intransitively).

For the output of `join_list`, `remove_incompletes` is called on the list of parses. `remove_incompletes` calls `is_complete` on each parse tree in the `tree list` and removes the trees for which `is_complete` returns false. This function `is_complete` calls `get_leaves` on its input tree to extract the list of leaf labels from the input tree, and then determines if any of the leaf labels belongs to the set of nonterminal symbols. If so, the tree is incomplete, and if not, the tree is complete since all leaves contain terminals. Therefore, `remove_incompletes` calls this function on each parse and removes the incomplete parses. The complete parses are then printed out.

In summation, this algorithm is able to take in a list of lists of trees, where each list of trees contains trees that are anchored by a given word in an input sentence. It finds a tree which is formed by combining, via TAG operations, one tree from each `tree list` in the `tree list list`. That tree must also be complete, and the lexical items of its leaves from left to right must be equivalent to the original sentence. This is the essence

of the parsing algorithm. All that remains is to take an input sentence and produce the `tree list list` input for `join_list` by finding all the elementary trees anchored by each word in the input sentence.

4.7 Base Cases: Elementary Trees

This next section will describe the process by which the `tree list list` mentioned above is generated. We will begin with all of the functions named after nonterminal symbols (i.e. `vp`, `dp` and so on). These functions take in a word (`string` lex) and output a list of elementary trees anchored by that word. For example, calling the `dp` function creates a list containing the basic determiner D and the determiner phrase DP, both of which are anchored by a determiner (in this case the input 'lex' is that determiner). Note that some of these functions do not have a `string` input but rather a unit input– `wh_ex`, `tc_cxn`, `wh_q` etc. These are all filler gaps, which as I have mentioned before are all anchored by the 'GAP' null lexeme and not an actual vocabulary word.

I have defined a limited vocabulary for testing purposes, represented in a hashtable. Each word in the vocabulary is a key in the hashtable, and the corresponding value determines which elementary trees that word can instantiate. For example, if a word has the corresponding hashtable value VP then all the elementary trees it can instantiate are generated from calling `vp` on the word. Many words require multiple function calls to get all the elementary trees. For example, the word 'to' may belong to a prepositional phrase ('I went to the store') or to an infinitive ('I like to run'). Many of the question words/ wh-words require multiple function calls as well– for example, 'what' can instantiate multiple types of DP as well as NP. 'Which' can instantiate an NP, a DP, or an adjunct relative clause (recall from the prior paper that relative clauses are adjuncts with head and foot in DP, to modify DPs i.e. 'the child **who I saw** was running'). Words of saying, knowing, thinking etc ('IndS') can instantiate a simple transitive or intransitive VP or an indirect discourse

(VP with S object rather than DP object). Ultimately, defining all the elementary trees that can be anchored by a given word is a very brute-force/hard-coded process which is taken directly from the linguistic research. The assignment of elementary trees is performed in the `get_pos` function which simply contains a series of if/else statements for each type of value in the hashtable, determining which function calls need to be made to generate all the elementary trees which can be anchored by the input word.

It should also be noted that the elementary tree generating functions for the filler gaps are highly dependent upon the `f_compare` function. As noted in the previous paper, defining each type of filler gap construction is essentially a meticulous process of noting down which combinations of filler, main clause, and gap can be used for each construction. For example, the Wh-relative can take who, whom, which, or that NPs, or a whose DP in the filler with a gap in the DP complement. Therefore, `f_compare` allows substitution of all those nonterminal symbols into the Wh-relative filler with a gap in the complement. Meanwhile Wh-relatives can take when, where, and why PPs in the filler with a gap in the PP adjunct. Therefore, `f_compare` allows substitution of all of those nonterminal symbols into the Wh-relative filler with a gap in the adjunct. Recall from the previous paper that filler and gap must agree– if the filler contains an adjunct, the gap must also be in an adjunct, if the filler contains a complement, the gap must also be in a complement. Therefore, `f_compare` helps specify which fillers can be substituted into elementary trees anchored by a gap in the adjunct (since the filler must also be an adjunct), vs which fillers can be substituted into elementary trees anchored by a gap in the complement (since the filler must also be a complement).

The function `run` essentially puts this entire process together. First, it takes the input sentence and breaks it into a list of strings (where each element of the list is a word in the input sentence) by calling `String.split_on_char`. Then, it traverses that list of words and calls `get_pos` on each one to produce a `tree list` of elementary trees anchored

by that word. It combines all these `tree lists` into one `tree list list` headed by a designated start symbol (S in the definition of a TAG in the previous chapter). Note that since we are parsing sentences, the linear sentence S is the designated start symbol. Recall from the previous paper that topicalized clauses, Wh-exclamatives, and indirect questions (wh-int-dep) can all stand on their own as complete sentences. Therefore, they are valid replacements for the designated start symbol S. So the `run` function attempts to parse the sentence as a linear sentence, or any other type of clause that can stand on its own as a complete sentence. It feeds each of these into `join_list`. As described above, `join_list` can then use the TAG operations to combine the elementary trees into a complete tree that represents the full sentence, parsed in according to CxG.

5 Results and Analysis

One of the initial goals of this project was to evaluate the practical applications of the TAG formalism for analyzing sentences using Construction Grammar theory. That is, this project set out to answer the questions of (1) can a parser be built using an implementation of TAG operations to parse a sentence into its component constructions? and (2) does this Construction Grammar based parse eliminate the issues with parsing filler-gap sentences using wh-movement? In this chapter, I will aim to answer those questions.

5.1 Evaluating the Parser on Correct Sentences

As I mentioned in the formalism chapter, the TAG-CxG grammar (or any ideal grammar) for a language should be able to generate every correct sentence in a language, and should never generate an incorrect sentence. In this first section, I test the TAG parser on grammatical English sentences to ensure that the parser is not undergenerating– that is, to make sure that every grammatical filler-gap sentence can be produced from TAG operations on the elementary trees for the filler-gap constructions. In process of writing

sentences on which to test the parser, I made sure to include all of the different categories of filler/gap and main clause. I also included various additional complements and adjuncts (including using Wh-relatives as DP adjuncts) into the sentences as well to demonstrate that the parser preserves the generative aspect of language as well and is not simply substituting into a pre-determined set of categories.

The test cases for topicalized clauses were:

(5.1) The good child, the student liked (i.e. but the bad child, the student did not).

(5.2) The good child, the mother knew the student liked.

(5.3) Quickly, the student ran.

(5.4) Quickly, the student who the child saw ran.

(5.5) In the store, the child ran quickly.

Examples 1 and 2 demonstrate a complement (DP) filler, with example 2 also including an additional auxiliary phrase 'the mother knew'. Examples 3, 4, and 5 demonstrate adjuNct fillers (AdvP and PP), with example 4 also including an auxiliary wh-relative and example 5 including an auxiliary adverb phrase.

The test cases for the Wh-exclamative were:

(5.6) What a good book the student read!

(5.7) What good books the student read quickly!

(5.8) How quickly the mother knew the child read!

(5.9) How quickly the child read the book!

Examples 6 and 7 demonstrate a complement filler (What-a DP and What-pl DP respectively). Example 7 additionally includes an auxiliary adverb phrase. Examples 8 and 9

demonstrate adjunct fillers (How AdvP), with example 8 also including an auxiliary indirect statement 'the mother knew' and example 9 including a complement to the main verb, the DP 'the book'.

The test cases for the Wh- direct question were:

(5.10) What book did the good child read quickly?

(5.11) What book did the mother know the good child read quickly?

(5.12) What did the child see?

(5.13) Whom did the child whom the student knew see?

(5.14) Why did the child run to the store quickly?

(5.15) When did the good child run to the store?

(5.16) Where did the child whom the student knew run?

(5.17) Whose mother did the student see in the store?

(5.18) Who did the child see in the store?

(5.19) What book did the child, whose mother the student knew, read?

(5.20) In which store did the child run?

(5.21) How quickly did the child run?

These examples cover all the different complement fillers (Who, Whom, What NPs, Whose and What DPs) and adjunct fillers (when, where, why PPs, wh-PP, How AdvP). They also demonstrate the adjunction of different types of auxiliaries including adjectives (10, 11, 15), adverbs (10, 11, 14), Wh-relatives (12, 16, 19), prepositional phrases (14, 15, 17, 18), and indirect statement auxiliaries (11).

The test cases for Wh- indirect questions were:

(5.22) The student knew the child knew how quickly the boy ran.

(5.23) The girl knew what the good child saw in the store.

(5.24) The boy knew why the child ran to the store quickly.

(5.25) The boy whom the girl saw knew what book the student read.

(5.26) The boy whose mother the girl saw knew when the child ran to the store.

(5.27) The good boy knew where the girl ran.

(5.28) The boy knew where to run.

(5.29) The boy knew who to see in the store.

(5.30) The boy knew whose child the girl saw in the store.

(5.31) The girl knew how quickly to run to the store.

(5.32) The girl knew in which store to see the child.

(5.33) The girl knew what books to read.

(5.34) The student knew whose mother to see.

These examples demonstrate all the different complement fillers (Who, Whom, What NPs, Whose and What DPs) and adjunct fillers (when, where, why PPs, wh-PP, How AdvP), as well as both the main clause types (infinitive and linear sentence). They also include auxiliaries such as adjectives, adverbs, prepositional phrases, Wh-relatives, and indirect statement auxiliaries just as the previous examples did.

Finally, these were the test cases for the Wh- relatives:

(5.35) The boy who the child knew ran.

(5.36) The boy saw the book which the child read quickly.

(5.37) The girl whose mother the boy knew read a book.

(5.38) The store in which a child ran opened.

(5.39) The girl whom the boy knew read a book quickly.

(5.40) The boy read the book that the girl read in the store.

(5.41) The store in which to run opened.

(5.42) The girl that the boy knew saw the student.

These examples demonstrate all the different complement fillers (who, whom, which, that NPs, whose DPs) and adjunct fillers (Wh-PP), as well as the different types of main clauses (linear sentence with either type of filler, infinitive with the PP filler). They also include auxiliary adjectives, adverbs, prepositional phrases, and indirect statements, and DP complements (37, 39).

These examples seem to suggest that this parser is effective for generating a Construction Grammar syntax tree for grammatical sentences and does not directly suffer from issues of undergeneration. However, looking at the actual code for this implementation, there is a clear scalability issue. While this small case study of filler-gap sentences does not suffer from undergeneration within that specific context, there are millions of types of elementary trees in a natural language. Developing a general parser for all types of speech would be an enormous task for an individual to do by hand, since a function would have to be written to generate all of those elementary tree base cases. This process of generating elementary trees would likely need to be automated by some statistical/machine learning algorithm and then later verified by native speakers to be feasible on a larger scale.

5.2 Evaluating the Parser on Incorrect Sentences

Although it is of course crucial that the parser should generate all grammatical sentences, it is equally important that it should not generate ungrammatical sentences. In fact, Construction Grammar's success in avoiding ungrammatical sentences is, from a theoreti-

cal perspective, its most attractive feature in comparison to UG-based grammars (which often generate ungrammatical sentences, particularly in the realm of Wh-movement). Therefore, the true measure of this TAG-based parser's success in implementing Construction Grammar would be its ability to reject ungrammatical sentences.

Therefore, I generated several ungrammatical test cases, drawn again from the previous paper, which many Wh-movement based analyses tend to accept. In particular, many of these examples involve 'moving' the filler out of a Wh-island, which is a location in the sentence from which Wh-movement cannot occur. While the parser rejects the majority of ungrammatical sentences, it does accept a few ungrammatical sentences, in particular those associated with a Wh-clause itself serving as a Wh-island.

These were the examples that I used:

(5.43) *The store what the child saw opened.

(5.44) *The speed how quickly the girl ran increased.

(5.45) *The book did the girl read?

(5.46) *The boy whose the girl saw mother read a book.

(5.47) *Where did the girl know who ran?

(5.48) *Who did the girl know when the boy saw?

For example 43, Wh-movement would allow the extraction of a what-DP from the complement of opened, but CxG does not allow any what-phrases in the filler of a relative clause. For example 44, Wh-movement allows the extraction of a How AdvP from the adjunct of ran, but CxG does not allow any how-phrases in the filler of a relative clause. In example 45, Wh-movement allows the extraction of the complement 'the book' from the complement of 'read' (i.e. like in 'the book, the girl read') but CxG prevents a topicalized

clause from having an inverted sentence as the main clause. Example 46 shows extraction from a Wh-island, the modifier of a complement 'whose'. CxG does not generate sentence 46 because the GAP must in the complement or adjunct of the verb, and 'whose' is an adjunct of 'mother' instead. Example 47 shows extraction from a a clause which has already undergone Wh-movement, which is another Wh-island. CxG does not generate sentence 47 because 'know' requires a linear sentence or indirect question complement. Example 48 represents extraction from a Wh-clause, another Wh-island. Unfortunately, the parser still produces an ungrammatical parse on this sentence. It seems like this is an error of implementation rather than an error of theory– as demonstrated in the previous paper with a similar example, the CxG approach should not accept sentence 48 since 'when the boy saw' cannot be the main clause of a filler-gap (in which the GAP would be located). Main clauses for filler-gap sentences are linear sentences, inverted sentences, or infinitives, but 'when the boy saw' is itself a Wh-phrase and cannot be any of the correct categories for the main clause. A parser which entirely accurately represents the CxG parse of a filler-gap sentence should not accept 'when the boy saw' as the main clause, so this seems to be an error of implementation.

Ultimately, this TAG parser seems to do fairly well in avoiding overgeneration of ungrammatical sentences, but, as the final example shows, it does not entirely eliminate overgeneration. This suggests that further work must be done on the parser to ensure that it completely accurately captures this construction. We might therefore conclude that this TAG implementation is close to, but not quite equivalent to, the CxG description of filler-gap sentences presented in the previous paper.

5.3 Future Applications

One of the goals of producing a parser was to consider the practical applications of such a tool. In particular, a parser which generates syntax trees according to the rules of Con-

struction Grammar, rather than any other more common grammatical system, could help provide unique insights into research at the syntactic-semantic interface. For example, UG-based parsers are often used to analyze the syntactic position or semantic context in which specific words or syntactic structures are used but not others. These parsers can also be used to determine whether native speakers are more likely to use specific words or syntactic structures in certain semantic contexts than second language learners, or vice versa. This type of research generally depends on having a pre-parsed corpus such as the Penn Treebank, on which to perform the statisical analysis. Having a parser that parses by Construction Grammar instead might provide some alternative perspectives on this kind of research, especially because Construction Grammar is a lot more specific in its syntactic categories and structures than most UG based grammars are. So while it might not be obvious that certain words or structures are being used in interesting contexts in a UG based analysis, such patterns may become evident in a CxG based analysis. In order to perform these analyses though, corpora need to be parsed using CxG, which necessitates a more sophisticated version of the parser produced in this project.

6 Conclusion

Ultimately, the goals of this project were to motivate the pairing of Construction Grammar with the TAG formalism from a mathematical perspective, and to determine whether a TAG-based parser that accurately captures the Construction Grammar syntax trees could be built for practical usage. Chapter 3 provides the mathematical justification for using TAG to represent Construction Grammar and more rigorously defines the process by which grammatical trees are generated. Chapter 4 explains the parsing algorithm that uses the TAG structures and operations to parse sentences as designated by the Construction Grammar approach to filler-gaps presented in the previous paper. Chapter 5 evaluates the parser, determining that it generates grammatical filler-gaps as expected and does not undergenerate in this context, but also that it is limited by the amount of

hard-coding needed to define all the elementary trees and therefore faces obstacles for larger scale projects. Chapter 5 also concludes that this parser can avoid some– but not all– of the common instances of overgeneration that Wh-movement based parsing produces. However, this seems to be an issue with the implementation of the parser, and not the theoretical CxG framework, because as demonstrated in the previous project, the CxG framework should theoretically be able to restrict those particular parses. Finally, this paper concludes that a more sophisticated version of this parser could be a useful tool for statistical research at the syntactic-semantic interface, since Construction Grammar provides a unique perspective on syntax, but parsed corpora following the rules of Construction Grammar are, at this point, extremely lacking.

7 References

Alonso, M. (1999). Tabular Algorithms for TAG Parsing. Proceedings of the ninth conference on European chapter of the Association for Computational Linguistics, 1999. https://doi.org/10.3115/977035.977056.

Beuls, K. (2012). Grammatical error diagnosis in fluid construction grammar: a case study in L2 Spanish verb morphology. Computer Assisted Language Learning, 27(3), 246–260. doi: 10.1080/09588221.2012.724426

Bleam, T. et. al. (2001). A Lexicalized tree adjoining grammar for English. Philadelphia, PA: University of Pennsylvania, School of Engineering and Applied Science, Dept. of Computer and Information Science.

Blumstein, S. E., Byma, G., Kurowski, K., Hourihan, J., Brown, T., Hutchinson, A. (1998). On-Line Processing of Filler–Gap Constructions in Aphasia. Brain and Language, 61(2), 149–168. doi: 10.1006/brln.1997.1839

Champollion, L. (2007). The XTAG Project. https://www.cis.upenn.edu/ xtag/.

Ciortuz, L., Saveluc, V. (2012). Fluid Construction Grammar and Feature Constraint Logics. Computational Issues in Fluid Construction Grammar Lecture Notes in Computer Science, 289–311. doi: 10.1007/978-3-642-34120-5-12

Croft, W. (2001). Radical Construction Grammar: Syntactic Theory in Typological Perspective. New York: Oxford University Press.

Goldberg, A. E. (2013). Constructionist Approaches. In The Oxford Handbook of Construction Grammar, edited by Thomas Hoffmann and Graeme Trousdale.

Hoffmann, T. (2013). Abstract Phrasal and Clausal Constructions. In The Oxford Handbook of Construction Grammar, edited by Thomas Hoffmann and Graeme Trousdale. DOI 10.1093/oxfordhb/9780195396683.013.0002.

Kroch, A., Joshi, A. (1985). The Linguistic Relevance of Tree Adjoining Grammar. University of Pennsylvania Department of Computer and Information Science Technical Report No. MS-CIS-85-16.

Lichte, T., Kallmeyer, L. (2017). Tree-Adjoining Grammar: A Tree-Based Constructionist Grammar Framework for Natural Language Understanding. In AAAI 2017 Spring Symposium on Computational Construction Grammar and Natural Language (Palo Alto, CA, 2017)

Nesson R., Shieber, S. M. (2006). Simpler TAG semantics through synchronization. In Proceedings of the 11th Conference on Formal Grammar (Malaga, Spain, 29-30 July 2006).

Sag, I. A. (2010). English Filler-gap constructions. Language 86, no 3: 486-545.

Steels, L. (2011). A design pattern for phrasal constructions. Constructional Approaches to Language Design Patterns in Fluid Construction Grammar, 71–114. doi: 10.1075/cal.11.06ste.

Storoshenko, D. R., Han, C.-H. (2013). Using synchronous tree adjoining grammar to model the typology of bound variable pronouns. Journal of Logic and Computation, 25(2), 371–403. doi: 10.1093/logcom/exs064

Trijp, R. V. (2015). Cognitive vs. generative construction grammar: The case of coercion and argument structure. Cognitive Linguistics, 26(4). doi: 10.1515/cog-2014-0074

Trijp, R. V. (2019). How a Construction Grammar account solves the auxiliary controversy. Benjamins Current Topics Case Studies in Fluid Construction Grammar, 79–104.

Vijay-Shanker, K. (1992). Using descriptions of trees in a tree adjoining grammar. Computational Linguistics 18 no. 4: 481-517. https://www.aclweb.org/anthology/J92-4004