INTRODUCTION

In *Parsing and Empty Nodes*, Mark Johnson and Martin Kay (1994) investigate the problem of parsing natural language according to grammars that allow for empty nodes. The existence of empty nodes, which represent traces left behind by moved constituents, often poses problems for standard parsing algorithms.
Standard parsing algorithms are comprised of successive steps, each of which proceeds by parsing elements of an input string consecutively. In order for an algorithm to be successful, it must finish processing input with the correct parse tree as the result.
THE PROBLEM

Although the words in the input string presented map to lexical items used for parsing, empty nodes, which are crucial elements of the parse tree, are not overt, making them difficult for the parser to detect. Unfortunately, parsers rely on an exact representation of all items in a sentence to give a correct parse, with little room for leeway.
THE PROBLEM

If an empty node is not detected, or too many are detected, the algorithm will give no parse for the sentence. Such an outcome is obviously not desired for parsers which have been assumed to handle natural language, but worse, an incorrect representation of empty nodes can cause the algorithm to fail in two ways. In the less severe case, the parser simply stops, failing to give a grammatical account for a sentence that should have one. More tragic, however, is the case in which the process cannot terminate, as it tries without fail to add empty nodes to lead to a parse.
THE PROBLEM

Thus, it becomes crucial to use overt lexical items of the sentence to correctly determine the properties of its empty nodes, giving not only an upper bound on the number that can occur (to prevent an infinite loop and ensure termination), but better, to provide a way for a parser to validate situations in which it expects an empty node to occur.
Consider the following sentence:

(1) What did John eat _e_?

This sentence is an example of wh-movement in English, in which the direct object wh-phrase moves to the front. Normally, the direct object would occur after eat. In order to preserve this relationship there must be an empty node after eat which is linked to the wh-phrase.
While it seems simple workarounds could be implemented for the parser to perform correctly in this situation, not every sentence, especially not those occurring in discourse, will have such a simple structure. Even worse, wh-movement can be long-distance, leaving several intermediate empty nodes as the phrase moves to the front of the sentence:

(2) What do you think e John said e that he ate e?
Furthermore, empty nodes will occur in more situations than just wh-movement. They occur in situations of extraposition

(3) I saw a book e at the store that you’d really like.

and coordinate structures

(4) John likes bananas and Mary e apples.
The situation is not as simple as it may seem on the surface. However, Kay and Johnson’s suggestion of using overt lexical items in the input string to “sponsor” a possible empty node aid in the tractability of this problem. Applied to standard parsers, the technique replaces bare lexical items with extended lexical items, retrieved during lexical access. An extended lexical item includes a non-empty lexical item, plus all of the empty nodes that it sponsors.
SPONSORSHIP

During the derivation, the parser first considers possible trees of empty nodes that are all sponsored by items in the processed input. Then, it generates rules to incorporate these trees containing empty nodes into the parse tree proper. The sponsorship properties of the non-empty nodes are then used to check the validity of the derivation step by step.
The problem now is to determine how sponsorship is assigned to lexical items. Kay and Johnson give an account, based on their observations of different phenomena, on how to assign sponsorship in very specific situations. However, none of these assignments are linguistically based, and have nothing to do with the idea of government of empty traces of movement. Moreover, the syntactic trees the authors in the paper are, in most cases, incorrectly derived, undermining their notion of sponsorship and its ability to aid in parsing.
AN ALTERNATIVE

In Kay (forthcoming), I propose an alternative, linguistically-oriented parsing algorithm that successfully handles the problem of empty nodes. Grounded in the same transformational grammatical principles by which empty nodes are licensed to exist, it generates a parse tree at deep structure and uses principles to invoke transformations that match the word order of the input string.
AN ALTERNATIVE

Based on the idea that all empty nodes represent the base position of a constituent, a generation of parse trees at both deep structure and surface structure determine the exact structural location of every empty node.
AN ALTERNATIVE

Thus, the parser has no notion of sponsorship as outlined by Johnson and Kay (1994). Instead of choosing an arbitrary lexical element from the input string to license the existence of an empty node, it is moved constituents themselves that license their traces.
AN ALTERNATIVE

The fact that the algorithm is linguistically based is crucial in its proposed success. Parsers of the type mentioned in the paper have no knowledge of the transformation properties outlined in modern theories of generative grammar. Due to this deficiency, the structural situations in which empty nodes can appear are essentially unbounded.
AN ALTERNATIVE

Though this algorithm differs from those of previous parsers in the amount of linguistic knowledge it relies on, it is entirely dependent on an accurate part-of-speech tagging algorithm that is run on the input string. The ambiguities that arise at this initial process are beyond the scope of this algorithm, so an accurate POS-tagging process is assumed for successful application.
Once each element in the input string has been successfully tagged for its part of speech, the algorithm proceeds in an entirely bottom-up, compositional manner to generate several candidate parse trees for the sentence’s deep structure. The algorithm invokes movement operations on each candidate resulting from properties of case assignment, \( \theta \)-role assignment, and other syntactic principles, to produce a surface structure tree that matches the word order of the input string. Only the candidate that leads to this proper final tree is considered to be the correct deep structure tree.
THE ALGORITHM

The first step in this process is to ensure proper tokenization of the input string. While for the most part each lexical item in the parse tree will map to a single word in the input string, there are exceptions involving inflectional morphology, null items, and force properties. The algorithm must have working knowledge of how to distinguish these elements, provided by a human-annotated index, in order for tokenization to succeed.
THE ALGORITHM

Once the input string has been properly tokenized and separated into distinct lexical items, it associates with each item a simple, unprojected syntactic tree whose root corresponds to the item’s part of speech. Once these trees are generated, each tree is successively projected upwards, adding complement and specifier positions along the way based on the argument structure of the lexical item. Again, a comprehensive index of the argument structure associated with each lexical item must be provided.
THE ALGORITHM

After all trees have been maximally projected to their full forms, composition of a deep-structure parse tree begins. Each tree is analyzed for any newly formed specifier and complement positions, which in most cases candidates for substitution. For each of these positions in a tree, the algorithm chooses another tree with a matching root node at random to substitute.
THE ALGORITHM

The composition proceeds, gradually decreasing the number of available trees, until all substitution positions have been filled and a single parse tree results. This parse tree is then a candidate for the correct deep-structure tree associated with the sentence. This process repeats, starting again from the project trees of the lexical items, to form all possible combinations of composition. This collection now constitutes all deep-structure tree candidates, only one of which is correct.
THE ALGORITHM

At this point in the algorithm, a collection of transformational rules is applied to each candidate deep-structure tree, taken from the most recent knowledge of syntactic theory. In essence, from each rule the configuration of the tree will be checked; if a condition is violated, the process invokes necessary movement operations. Once all operations have been completed on a candidate tree, we have arrived at the single surface-structure tree for this tree.
THE ALGORITHM

The tree collection now includes $n$ candidate deep-structure trees, and $n$ candidate surface-structure trees, in a one-to-one correspondence. We then check the word order of each surface-structure tree candidate, extracting each word using a depth-first search. If the sentence has a specified force, this consideration is made as well. If the correct word order and force properties are found for a tree, then its parse tree has been found, with any empty nodes in their proper positions.
What follows are step-by-step executions of the algorithm on various input strings to demonstrate its success. Consider first a simple sentence:

(1) John eats bananas.
First, the sentence is properly tokenized into a collection of lexical items:

(2) John eat -s (some) bananas

Note that while three words appear in the input string, the sentence itself is composed of five lexical items. *Eats* is a morphological composition of the verb *eat* and the inflectional present tense singular morpheme -s, while *bananas* is a plural noun and thus implies a covert determiner, *(some)*. The process by which the parser detects this bipartite structure is straightforward.
DEMONSTRATION

To correctly deduce inflectional morphology, the algorithm needs to determine the most probable part of speech for each word. Through use of either annotated corpora or a simple lexicon index, it is straightforward in this example to determine that John is a proper noun, eats is a verb, and bananas is a plural noun.
Since *eats* is now known as a verb, the parser then detects the inflectional element -s and separates it out as a lexical item, making sure to check the validity of this extraction through phonological principles. The word *bananas*, however, being a noun, does not receive this same analysis.
Instead, the status of *bananas* as a plural noun causes the parser to posit a null determiner, *(some)*, which precedes it. In other languages (as well as other dialects of English) in which other determiners are null, this process works as well (consider the AAVE null possessive determiner).
We now have the correct collection of lexical items for this sentence, repeated below:

(2) John eat -s (some) bananas

Making use of POS-annotated corpora, the algorithm associates with each lexical item its most probable part of speech. In this example, relatively few ambiguities exist. John, being a proper noun, is a determiner. Eat unambiguously is a verb, and its directly following present tense morpheme -s must be an inflection. (Some) is necessarily a determiner, and bananas, a noun.
DEMONSTRATION

Using standard abbreviations, we now have a correctly tokenized and identified collection of POS-annotated lexical items representative of the input string:

(3) [D John] [V eat] [I -s] [D (some)] [N bananas]
DEMONSTRATION

A collection of the lexical items represented in tree form is then the following:

D
V
I
D
N

John
eat
-s
(some)
bananas
DEMONSTRATION

The next step, the projection of these trees to allow for specifier and complement positions, is the first step that relies on linguistic knowledge encoded into the algorithm. Step-by-step, we will show how the trees are projected based on the supposed argument structure of the lexical items using theta-theory.
John, a proper noun, takes no arguments of its own, and simply projects up two levels: first to an intermediate projection (at a single bar level), and then to a maximal (phrasal) projection. The full projected tree is simply the following
where “DP” indicates “determiner phrase.” This is the full tree for the lexical item *John*, to be substituted.
DEMONSTRATION

The next lexical item, *eats*, is more complicated in that it assigns two \( \theta \)-roles, or argument positions: an agent, a role external to the meaning of the verb which specifies the entity performing the action, and a theme, which is internal and refers to the recipient of the action. (Other verbs, including intransitive verbs, assign only one \( \theta \)-role, in this case the agent, since there is no recipient.)
Armed with linguistic knowledge of how certain $\theta$-roles are assigned, the algorithm posits a specifier position for the entity receiving the agent $\theta$-role, and a complement position for the entity receiving the theme theta role. The specifier is the sister of $\overline{V}$, and the complement a sister to $V$ itself. As for the types of these positions, a common assumption is that both positions will be nouns, possibly with a determiner, so the posited category for both is DP.
Thus the proposed full projection for *eat* has the following structure

```
VP
  /
DP  V
  /
V  DP
  /
eat
```

with the agent in the leftmost, DP specifier position and the theme in the rightmost, DP complement position.
A similar analysis is provided for the inflectional particle -s. While not assigning theta roles itself, the tree must still project specifier and complement projections just like the verb. Essentially, the inflectional morpheme takes as an argument the verb to which it attaches, so it must have a VP complement. The Extended Projection Principle of Chomsky (1981), the details of which are beyond the scope of this presentation, necessitates the need for a specifier position in an inflectional phrase as well, which is a DP.
Mirroring that of *eat*, the full projection for -s is as follows:

```
<table>
<thead>
<tr>
<th>IP</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>DP</td>
</tr>
<tr>
<td>I</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>VP</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>-s</td>
</tr>
</tbody>
</table>
```
Moving on, the lexical item *(some)* is a simple determiner; straightforwardly, it must be complemented by the noun phrase that it modifies. No specifier position is necessary:
Finally, *bananas* is a simple noun, taking no arguments, and its projection is simple:
DEMONSTRATION

We now have the following collection of fully projected trees that will be composed to eventually form the deep-structure of the given sentence:

```
DP  VP  IP  DP  NP
  \  /  \  /  /  \\
D  DP  V  DP  N
  /  /  /  /  /  \\
D  V  DP  I  N
   /  /  /  /  /  \\
John eat -s (some) bananas
```
Now at the next major step of the algorithm, numerous candidates for a deep-structure parse tree of the sentence will be generated through composition of these lexicalized trees. As stated before, the algorithm proceeds through each tree, and for each tree, analyzes each node that requires substitution. Since the tree for John has no substitutable nodes, we will move on to the next tree, the one for eats, which needs both its specifier and complement positions filled.
DEMONSTRATION

To find a possible substitution tree, the algorithm simply determines the label of the node that needs to be substituted into, then picks any of the other trees at random with that label as the root.
DEMOnstration

[Diagram showing a tree structure for the sentence: John eats some bananas]
DEMONSTRATION

```
VP
   /\    
/   \   /
  DP   V  DP
     /   /
    D   V
     /
    eat

John

IP
   /\  
/   \ /
  DP  I
     / 
    I
     /
    -s

(some)

NP
   /\  
/   \ /
D   VP
   /   /
N   N

bananas
```
John eats (some) bananas.
John eat (some) bananas
John -s (some) bananas
DEMONSTRATION

[Diagram of a dependency tree with labels such as DP, VP, I, V, NP, and N, along with words like John, eat, (some), bananas.
Finally, various transformational rules are applied to arrive at a surface structure. The prescient rule in this case is the satisfaction of the EPP, which states that the specifier of IP position must be filled. Since this position must be a DP, we move the DP in the specifier of VP position to the specifier of IP position, fulfilling the EPP. Such movement leaves a trace, which is coindexed with the moved element.
John eat (some) bananas
Lastly, we apply a morphological rule that states that in English, the inflectional particle attaches to the verb when we encountering. Now, performing a depth-first search on this tree, we receive the proper word order, taking care not to include the implied determiner:

(5) John eats bananas.
The next example is the first demonstration of how the algorithm handles empty nodes. Consider the question counterpart of the sentence in (2):

(6a) What does John eat?

Such a sentence is an example of wh-movement in English, in which the theme argument, the direct object, moves to the left edge of the sentence. That is, it originates in the verb’s complement position, but due to a transformation moves to another position.
DEMONSTRATION

To see this movement, consider the counterpart of this sentence asked as an echo question, possibly by a speaker who is incredulous or thinks he may have misheard:

(6b) John ate what!?

In an ordinary declarative sentence, of course, the direct object follows the verb, as we saw from (2).
To indicate that movement has occurred, we say that the direct objet, in this case *what*, has left a trace in its original position, with which it is coindexed:

(6) What does John eat \( t_i \)?

This trace is then an empty node, an element that raises problems for most parsers. But the algorithm being described currently will determine the proper structural position for this empty node, even without determining another lexical item to sponsor it.
To demonstrate its success, we will work through a step-by-step application of the algorithm as before. Since the sentences are similar, many of the processes will proceed in the same way, the details of which should now be straightforward and we will omit. To start, we must properly tokenize the sentence; since it is interrogative, we must be careful to take into account the force of this sentence, which is a lexical item itself:

(7) what does John eat (?)
Part-of-speech tagging proceeds as before. *What*, being a wh-pronoun, is a determiner, and *does*, indicating tense, is an inflection. New here is the category for the interrogative force of the sentence, represented as (?). Based on syntactic theory, we will posit that it is a complementizer, like *that*:

\[ (8) \ [D \text{ what}] [I \text{ does}] [D \text{ John}] [V \text{ eat}] [C (?)] \]
DEMONSTRATION

In the projection process, *what* takes no arguments, leading to a simple tree, and the complementizer (?) takes both a specifier and a complement. Note, however, that unlike all other positions we’ve encountered so far, the specifier in this tree can remain unfilled at deep-structure, as long as it becomes filled at surface structure.
DEMONSTRATION

[Diagram of syntactic structure with labeled nodes: DP, CP, C, IP, D, what, (??)]
DEMONSTRATION

CP
  /\  
DP  C   IP
     /\  
   DP  D   DP  D
      /\            /\ 
     D  D          I  VP
        /\        /\ 
       D  D      DP  DP
          /\    /\  
         D  D   V  DP
            /\    /\  
           D  D  I  DP
              /\  
             D  what does eat
DEMONSTRATION

CP
  / \  
DP   C
     / \ 
(?)  DP  IP
     / \  / 
    I  I VP
     |  |  |
    does what John eat

DP  VP
  /  /  
D  D  D
  /  /  
D  D  DP
John does eat

DEMONSTRATION
John does eat what
Now having reached a candidate deep-structure parse tree, the first transformation to invoke is movement from the specifier of VP position to the specifier of IP position to fill the EPP, as before:
John does what eat.
Next, as is a property of wh-pronouns in English, *what* must move from the complement of VP position to the specifier of CP position:
DEMONSTRATION

what does John eat?
One remaining rule applies to bring this tree to surface structure. Unlike in other languages (for example, Irish), the (?) complementizer is phonologically null. In order for this element to have meaning, it must be phonologically realized by combining it with another element. Since we know that inflectional elements can move, and that heads can adjoin to others, we posit this movement as head adjunction from I to C:
DEMONSTRATION

```
CP
  /   \
/     \
DP_j   CP
  |     |
D     C
  |     |
D     IP
  |     |
D     I
     |
D     VP
     |
D     V
     |
John  
     |
t_k   
     |
DP    
     |
V     
     |
eat   
     |
t_i   
     |
DP    
     |
t_j   
```
DEMONSTRATION

Now that all of the transformational rules have applied, we check the tree to see if its word order and force match. Since they do, the algorithm has successfully determined the proper parse tree for the input string.

(8) What does John eat?
CONCLUSION

The examples given are simple, but the strength of the algorithm is that as long as there is a working theory in transformational grammar for a sentence in English, equipped with this knowledge the algorithm will be able to parse it. And since these theories apply to not just English but are universal, with parametric variation, across all languages, we have not bound the parser to a specific language and thus it is extremely robust.
CONCLUSION

With proper linguistic grounding, and extensibility to cover as many grammatical properties as necessary, any complication that may arise in linguistically uninformed parsers, including the problem of empty nodes, is covered, provided there is a theory for the derivation of sentences of that type.