Ambiguity in Minimalist Parsing:

Comparing E.Stabler's Minimalist Grammar to A.Weinberg's Minimalist Parser

Independent Study by
Ilia Kurenkov

Advisor
Brian Dillon

University of Massachusetts Amherst
April 28, 2013

Contents

1 Introduction

Chomsky's minimalist program inspired, among other endeavors, attempts at applying its tenets to modeling how humans process sentences. On the one hand there is Edward Stabler's Derivational Minimalism (Stabler 1997a). This approach is rooted in a tradition of formal grammars and automata well-studied in mathematics and computer science. As such, it is characterized by a very clear and detailed description of its elements and how they interact with one another. This type of grammar is conventionally referred to as Minimalist Grammars (MGs).

On the other hand, almost at the same time, a slightly different vision of a minimalist parser was proposed based on contemporary psycholinguistic theories of human sentence processing (Weinberg 1999). Certain principles generally accepted in the research community form the basis of this parser. In comparison with MGs these principles seem a bit less formally defined.

In this paper the relationship between these two models will be explored based on their performance on some linguistic data. In addition, some questions and observations will be fielded regarding the way both models interact with ambiguity.

The paper is structured as follows: a Minimalist Grammar and a Stabler-style parser/recognizer based on it will be defined in **Section 2** and **Section 3** respectively. **Section 4.1** will introduce Amy Weinberg's Minimalist Parser model and compare its properties to those of Stabler's MGs. In the section after that a piece of linguistic data from Weinberg's 1999 article will be presented along with her analysis of said data. Some concerns about the accurateness of this analysis will follow. **Section 6** features a brief overview of the problem of Lexical Ambiguity as well as one potential solution to it.

2 The Stabler Grammar

2.1 Formal Definition

According to (Stabler 1997a), a Minimalist Grammar (MG) is defined as a combination of a certain lexicon and a set of operations for generating expressions from this lexicon. For our purposes a simplified version of the MG will be defined, one that doesn't support Covert Movement[1].

A Minimalist Grammar is defined as a 4-tuple (V, Cat, Lex, F), where

$$\begin{aligned} V &= \{P \cup I\} \\ Cat &= \{base \cup selectors \cup licensee \cup licensors\} \\ Lex &= \{V \cup Cat\} \\ F &= \{merge, move\} \end{aligned}$$

The meaning of this notation is explained in the following sections.

2.2 The Features

V stands for Vocabulary and contains the non-syntactic parts of an expression. Every item in the Vocabulary has a phonetic and semantic (interpreted) component. They are respectively represented by P and I in the defintion of V. Phonetic features are conventionally placed inside slashes and the interpreted components inside parentheses. Thus the word *table* would be represented as **/table/(table)** in the Vocabulary. In this paper, however, these features will mostly be treated as a single unit[2] and rendered as the orthographic representation of the word they correspond to. The following two notations are thus equivalent:

(1) /table/(table)

(2) table

The syntactic features are specified in their own set, Cat (for Categories) which can be broken up into several subsets that distinguish between different types of features. Table 1 gives some examples of each type and the sets are described below.

[1] Covert and Head Movements are problematic for several reasons and not very relevant to the topic at hand. A detailed discussion of these issues can be found in (Harkema 2001) and (Michaelis 2001)

[2] This is due to the decision not to account for Covert Movement here as was mentioned above.

Table 1: Feature Sets

Set	Examples
base	v, d, c, n
selectors	=v, =d, =c, =n
licensors	+case, +wh
licensees	-case, -wh

$base$ consists of features that correspond to traditional syntactic categories, eg. verb, noun, adjective, complementizer and so on. For the sake of brevity a feature label is shortened to the first letter of the category it represents. For example, the feature set of the word *monkey* would contain a feature n because it is a noun.

selectors is based on *base* and can formally be defined as:

$$\{=\mathsf{x} \mid \mathsf{x} \in base\}$$

What this means is that for every syntactic category label (noun, verb etc.) there is a corresponding selector feature, represented as the equality sign concatenated with the feature label. When features from this set are encountered during derivation, the *Merge* operation is applied.

$licensors$ is the set of features that indicate that movement needs to happen. They are encoded in the form "+featurename" and represent such known syntactic movement drivers as case, tense and wh-phrase features. $licensees$ is the corresponding set of features that allow expressions to move. This set is similar to $selectors$ in the sense that it can be formally defined with respect to its complement set, like so:

$$\{+\mathsf{x} \mid \mathsf{x} \in licensors\}$$

As this definition suggests, an entry in $licensees$ takes on the form of "-featurename". The features in $licensors$ and $licensees$ are somewhat akin to Chomsky's φ*-features*.

2.3 The Lexicon

Lex is the Lexicon, it combines the syntactic features with the phonetic and interpreted (semantic) ones. Consequently, every entry in the Lexicon will contain zero or more members of the sets described above. In fact, Stabler posits the

following generalized lexical entry[3]:

$$selectors^*(licensors)selectors^*(base)licensees^*P^*I^*$$

This sequence of features already demonstrates some interesting properties of MG derivations. For example, an item cannot ever be the head of a phrase (be a selector or licensor) once it has itself been selected once. The same is true for a moving item: by the time an it can move whatever selector features were present to begin with would have been exhausted. An item also cannot be selected more than once. It can, however, move more that once. Lastly, new constituents can only be introduced via merger. The *Move* operation can only modify existing tree structures.

It is crucial that there may be zero members of any of these sets. This allows entries without any phonetic or semantic features to be part of the Lexicon. Some of these – strictly speaking – non-lexical entries correspond such syntactic notions as the phonetically null "C" and "T" in the "CP" and "TP" respectively. In an MG the null "C" and "T" can be represented this way:

=v c

=v +tense t

It is assumed that all such entries lacking a phonetic feature are accessible to the grammar at parsing time, whereas entries tied to input tokens have to be looked up in the Lexicon when the corresponding input is being processed.

Phonetically null entries do not represent the majority of the members of the Lexicon, however. Most entries in it are the ones whose phonetic and interpreted features are present. A possible lexical entry for the word *monkey*, for instance, could be:

n -case /monkey/(monkey)

This entry means the following:

- it can be selected by some other expression with the =n feature
- it can move if c-commanded by another expression with the feature +case
- it corresponds to an utterance that sounds like /mʌŋki/
- it loosely[4] refers to members of the order Primates

[3] The notation used means the following:
set^* = one or more members of set
(set) = zero or one members of set

[4] According to Merriam-Webster, 'monkey' does not include humans, tarsiers and lemurs and sometimes also excludes apes.

Lexical entries are also called "simple trees" because they consist of only one node. They serve as input to the structure building operations defined below. The type of operation applicable and its role in the result are determined by an entry's leftmost feature.

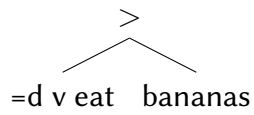
For example, the above entry for the verb *carry* first requires that a *Merge* operation combine it with some other expression, the leftmost feature of which is d. It is said that =d $\in$ *selectors* "triggers" *Merge*. The fact that it is a selector means the expression for *carry* will be the head of the phrase resulting from the application of *Merge*. Once it has been processed in this way, =d will be deleted from the expression and the next leftmost feature (here also =d) will become relevant. This incremental feature traversal/deletion continues until there are no features left to traverse/delete or until a root category (such as "c") reached.

2.4 Operations

The classic[5] MG defines only two structure building operations: *Merge* and *Move*. These operations are functions that accept expressions as input and return new expressions.

In the case of the *Merge* function a selecting head is combined with another expression as in (3).

(3) [=d =d v eat] + [d bananas] $\rightarrow$



The result of this combination is a complex (consisting of more than one node) tree with the head =d v eat and its argument bananas. The head of the phrase in the tree is conventionally indicated by an arrow pointing in its direction.

In a more general sense, if we have two trees t_1 and t_2 with leftmost features =x and x respectively, then

$$merge(t_1, t_2) = \begin{array}{c} < \\ t'_1 \quad t'_2 \end{array} \qquad \text{if } t_1 \text{ is simple} \tag{1}$$

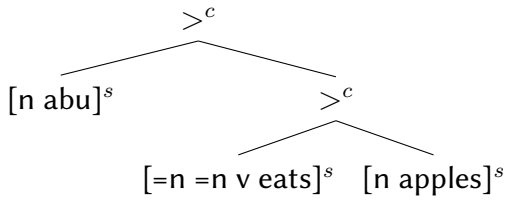
$$merge(t_1, t_2) = \begin{array}{c} > \\ t'_2 \quad t'_1 \end{array} \qquad \text{if } t_1 \text{ is complex} \tag{2}$$

[5] See (Frey and Gärtner 2002) for an example of an extension to MGs that adds two more operations: *adjoin* and *scramble*.

Where t'_1 and t'_2 are the same as t_1 and t_2 with their leftmost features deleted (checked).

Note that *Merge* returns slightly different items depending on whether the selecting (head) node is simple (lexical) or complex. This difference allows an MG to construct structures compatible with the X-bar Theory, where the complement is the internal argument that is generated on the right of its head and the specifier an external argument generated to the left of a head that already has a complement. The structure in 4 demonstrates this idea (the superscripts "c" and "s" for each node designate whether that node is complex or simple).

(4)



Positing such structures causes problems when naively analyzing intransitive VPs, because the overt subject (in specifier position) must combine with a complex phrase, not the lexical entry for the verb. In order for this to happen, something needs to occupy the complement position. A discussion of one possible account for intransitives can be found in Appendix A.

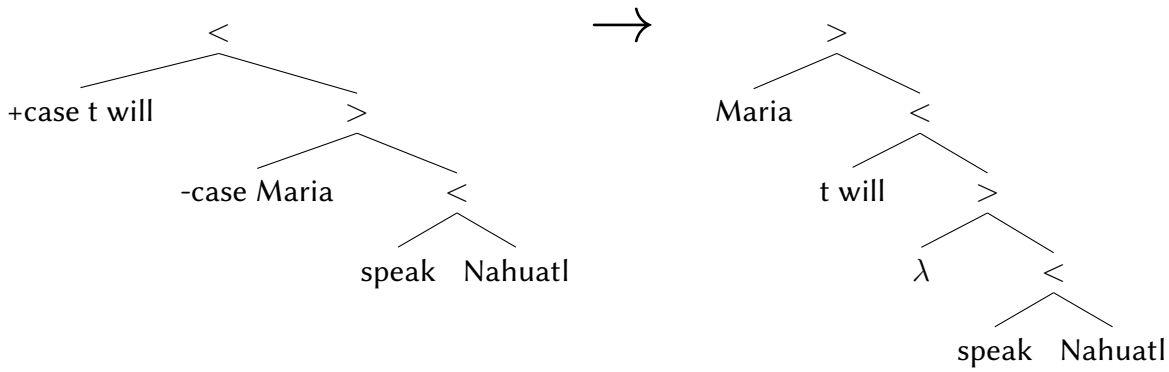
The function *Move* in contrast to *Merge* operates on only one input tree. It is triggered by the head of the tree having an active (leftmost) +f feature. It is important to note that movement is possible only if there is no more than one non-head phrase in the tree with the leftmost feature being -f. This restriction reflects the Shortest Movement Constraint similarly to how the structures created by *Merge* comply with the X-Bar Theory.

If we have a tree t that satisfies these conditions, then

$$move(t) = \begin{array}{c} > \\ \diagup\ \diagdown \\ t_1 \quad t_0 \end{array} \tag{3}$$

Where t_1 is the node anywhere inside t with the licensee feature that moves to specifier position and t_0 is the same as t but with an empty node (denoted as λ) instead of t_1.

Below is a slightly modified example of the operation applying borrowed from (Stabler 1997b).



2.5 Parsing Example

To demonstrate a Minimalist Grammar in action, let us consider how the sentence in (5) would be parsed, given the lexical entries in Table 2.

(5) abu stole the ruby

We start by combining "the" with "ruby":

[=n d the] + [n ruby] → d the ruby

=n d the n ruby

The result is then merged with "stole".

=d =d v stole + d the ruby → =d v stole the ruby

=d =d v stole d the ruby

After that the phrase "stole the ruby" is merged with "abu".

d abu + =d stole the ruby → v abu stole the ruby

d abu =d stole the ruby

Finally, all of this is combined with the C category.

=v c + v abu stole the ruby → c abu stole the ruby

=v c v abu stole the ruby

The "C" category is special. When it is reached, the structure is considered complete and no further derivation is necessary provided the grammar has reached

Word (token)	Feature Set
abu	d
stole	=d =d v
the	=n d
ruby	n
—	=v c

Table 2: Example Lexicon

the end of the input string. More will be said about such categories when the MG defined above is converted to a recognizer in the following section.

3 The Recognizer

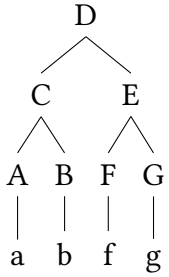
A recognizer is a type of parser that determines if a given input string of words is well-formed according to a certain grammar. In order to construct a recognizer based on an MG I will make use of a combination of two frameworks: Left-Corner Parsing and Deductive Recognizing.

3.1 Left-Corner Parsing

A parser takes in a sequence of symbols and then uses some set of rules and a lexicon to convert this sequence into a defined structured representation. A parser achieves this with a mix of (i) "reading in" subsequences of the input then generating corresponding structures from the Lexicon and (ii) applying rules to already constructed structures. Parsers differ with respect to what mix of input processing and rule application is used.

An example of an algorithm (strategy) heavily dependent on input is bottom-up parsing. Consider the structure in (6), where capital letters represent structures derived by rules and small letters stand for input strings. In order to construct the node C, a bottom-up parser must first process input strings "a" and "b", looking each one up in the lexicon and retrieving the corresponding entry for them. Not until both "A" and "B" have been created can the parser create node "C". By the same token the construction of the node 'D' requires that both nodes "C" and 'E' be created. Note that reaching node "D" signifies completion of the parsing of this input, since no further structure building operations are needed after this node is created. Therefore a bottom-up algorithm needs to process the entire string in order to construct the corresponding structure for it.

(6)



At the other end of the spectrum when it comes to input dependence is top-down parsing. This algorithm starts from "the top" of the tree, node "D" in the case of (6). Assuming that it has an appropriate rule for this, the algorithm then expands "D" into its daughters "C" and "E". They in turn are expanded each into its daughters and only when there are no more nodes to expand does the top-down parser start scanning the input to confirm its predictions about the structure it built.

In between these two extremes we find left-corner parsing[6]. This algorithm combines input dependence from bottom-up with the predictive tendency of top-down. It does this by stipulating that (i) a node can be created as soon as at least one of its daughters is considered fully recognized and (ii) a partially recognized node can be expanded into its daughters as long as it is indicated which ones are fully/partially/not recognized at time of expansion. A node is considered fully recognized if it is either (a) an entry in the Lexicon (both lexical and non-lexical) or (b) if all of its daughters have been fully recognized. A node can alternatively be marked as partially recognized if some (but not all) of its daughters are fully recognized. In all other cases a node is considered unrecognized.

Figure 1 shows how a simple implementation of a left-corner algorithm would traverse the structure in (6).

In iteration 1 the parser reads in "a" and finds the corresponding entry "A" in the lexicon. This is now a fully recognized item and can be used to build further structure. It then adds the mother of "A", "C", as a partially recognized node to the list[7] of Projected and Predicted Items. It is paired with "B", the yet unattested sister of "A".

With this we have exhausted the possibilities of projecting and predicting structure based on "A" and we also do not have any new fully recognized items.

[6] All these parsing algorithms can be thought of as instances of Generalized Left-Corner Parsing (Demers 1977) with different values for the index representing how many tokens of the input need to be recognized in order for the parser to start constructing a structure. In the case of bottom-up this index is set to the length of the string (complete input dependence), in top-down it is equal to 0 (complete independence from input) and in left-corner it is 1 (partial input dependence).

[7] In practice this is a priority queue. For more details, see Appendix **??**. In this paper the terms "stack", "list" and "queue" will be used interchangeably to refer to the same structure.

Iteration	Input	Fully Recognized Item	Projected and Predicted Items
1	a	A	[C B]
2	b	B	[C B]
3	None	C	[D E]
4	f	F	[E G], [D E]
5	g	G	[E G], [D E]
6	None	E	[D E]
7	None	D	None

Figure 1: Left-Corner Parsing Example

This means we read in the next substring and iteration 2 commences. The parser recognizes input string "b" as item "B". This new recognized item is compared to all the entries in Projected and Predicted Items, starting the most recently added ones. In this case, however, there is only one entry [C B]. The right side of this entry matches the current recognized item and consequently (i) that entry is removed from the unrecognized list and (ii) the item "C" is now fully recognized since it satisfied condition (b) mentioned earlier.

Thus begins iteration 3. It must be noted that the parser does not need to read in the next token from the string as long as there are fully recognized items left that have not been processed yet. The item "C" is used to generate the pair [D E] which is added to the list of projections/predictions. The parser then once again finds itself at a point where no unprocessed recognized items are left.

Iteration 4 starts with it reading in token "f", which leads to the recognition of item "F" and the projection/prediction of the pair [E G] (added, naturally, to the list). Iteration 5 introduces the token "g" and consequently the item "G". [E G] is recognized as a result and reduced from the stack. "E" is now the fully recognized item and is used in iteration 6 to reduce [D E] from the stack. We are left with the fully recognized item "D", which in our system signifies that the input sentence was successfully parsed and recognized.

Why was the left-corner algorithm described in such detail? There are strong indications that left-corner parsing is closest to whatever strategy humans use (Johnson-Laird 1983; Resnik 1992) because it shows the same ease/difficulty patterns as humans do when processing certain types of recursion[8]. Thus it will

[8] Johnson-Laird and Resnik discovered that both humans and left-corner parsers can handle left and right recursion without much trouble, but encounter difficulties with central embedding. For humans this difficulty is expressed in the inability to parse sentences like *The rat, that the cat, that the dog chased, bit, ate the cheese.* whereas for Left-Corner parsers the difficulty lies in the number of unrecognized items they need to keep in memory.

serve as the algorithm for the MG-based parser once some additional necessary machinery is defined that was only informally described above.

The projected and predicted items are stored in something we will call a *Queue* in this paper (it will sometimes also be called a stack). The reason for such a name is that this data structure is implemented like a priority queue and operates similarly to a stack. As the parser proceeds through the input string it uses the recognized items looked up in the Lexicon to generate pairs of semi-recognized + unrecognized items. Each pair is added to the *Queue* with a priority value from the set $\{x : x \in \mathbb{Z}, x > 0\}$, such that more recently processed pairs have higher priorities than the ones processed before them.

This is a way to represent the influence of recall in processing and predicts that if the features of a node being processed can be satisfied by several items, the closest one to the left in the string will be preferred consistently over candidates that occurred earlier in processing. Such a mechanism for the queue/stack is posited because it is in line with most research of recall and filler-gap dependencies.

An important exception to this mechanism are non-lexical (phonetically null) items. The *Queue* is initialized with all of their derivations present and no new ones are added during sentence processing. Moreover, all of them are added with equal priority 0.

The resulting preference for structures built from strictly lexical items as opposed to phonetically null categories such as "C" is a conscious design choice and reflects an attempt to prevent the parser from overusing the latter as well as to establish the primacy of actual input over purely syntactic constituents.

The interactions of the recognized items with the *Queue* are regulated by the *Scan* function. This function takes as input a fully recognized item α and the queue Q in some state and loops over the unrecognized items on Q, looking for a match with α. There are two possible outcomes of this search:

- If some matching unrecognized item β is found, both it and the corresponding semi-recognized item γ are removed from Q, after which the semi-recognized item γ (which is now fully recognized) is checked for the presence of an active licensor feature. If such a feature is present, the parser attempts to apply the operation *Unmove* (described below) to the item, otherwise the item remains the same. After this γ and Q are passed to an new iteration of *Scan*.

- if no match was found after all of Q was traversed, the arguments are returned as they are, unchanged.

The *Unmove* function corresponds to the function *Move* defined in **Section 2.4**. It takes as an argument an item α with an active licensor feature, eg. +case,

+tense etc in its head. It then looks at all the subtrees in α for one that has as active feature the licensee corresponding to the licensor feature of the head. If one (and only one) such subtree is found, it is moved to the specifier position of α, leaving behind an empty space (denoted as λ in **Section 2.4**). The result is then returned.

Returning to the *Scan* function, we find that once it has run its course there are no more unrecognized structures that can be confirmed by the current state of the input. The latest recognized item is now used to derive more structures that are to be added to the queue.

First, the *Up* function takes the fully recognized item and uses its features to project "upwards" in the tree structure. In other words, it creates the parent node of the current recognized item. If the active unchecked feature of the fully recognized node is a selector, indicating that the daughter is the head of the parent and dominates the other daughter, the parent node inherits the features of the recognized daughter. Otherwise the features are left blank, to be later inherited from the other daughter (presumably a head). The parent node is returned.

Next, the *Unmerge* function is used to predict the yet unseen sister of the recognized item passed to *Up* in the previous paragraph. It does this by looking at the leftmost unchecked feature of the recognized item and determining by that what the leftmost unchecked feature of the unseen item will be. The outcome of this function can be one of the following:

- If the unchecked feature f is a base feature, the item predicted must have an unchecked feature =f. Moreover, it must be a complex item, because the confirmed item is to the left of it[9].

- If the unchecked feature =f is a selector feature, the item predicted must have an unchecked feature f. Both complex and simple items are allowed, since they will occupy the complement position and there are no phrase type restrictions on that in the definition of *Merge*.

Note that only one feature is relevant at this stage of the derivation. The features following it will vary depending on what recognized item will ultimately match the item posited by *Unmerge* and thus do not need to be specified.

One last component remains to be defined: the categories that lie outside of the domain of *Unmerge*, the "root nodes" that the parser should not attempt to

[9]The reasoning behind this warrants some elaboration. As was mentioned in the description of the *Merge* operation in **Section 2.4**, the specifier position is always to the left of the head of the phrase just like the complement position always to the right. This means if during parsing we encounter a selectee before its selector, it is safe to assume that it will be in specifier position of whatever phrase selects it. According to the definition of *Merge* such a selector phrase must be complex.

expand. Such nodes are referred to as *distinguished categories* in some descriptions of MGs (Harkema 2001). They consists of categories such as "C" from (5) and "D" from (6). These categories are stored in the Lexicon along with other *undistinguished* categories. The parser must, however, have access to a list of all such categories so that it does not apply the *Up* and *Unmerge* functions to them.

This concludes the definition of an MG-based parser. In combination with the notion of deductive reasoning it will then form the basis for the recognizer.

3.2 From Parsing to Recognizing

In order to turn our parser into a recognizer the notion of recognizing strings and nodes introduced in the preceding section will be augmented by an insight that judging grammaticality is akin to deductive reasoning. This idea was first expressed in (Shieber, Schabes, and Pereira 1995) and to date most thoroughly applied to MGs by Harkema 2001.

In deductive reasoning one strives to prove that a certain set of premises will lead to a certain consequence by a series of inference steps determined by certain rules. This can be demonstrated on a classic introductory logic example of a conditional. Given a set of premises:

$$P \tag{4}$$

$$P \rightarrow Q \tag{5}$$

And a rule in the form[10]

$$\frac{\alpha \qquad \alpha \rightarrow \beta}{\beta}$$

One can apply the rule to the premises and derive the following conclusion:

$$Q$$

The reader will no doubt notice that this resembles closely how new nodes are created by a parser, be it top-down expansion or bottom-up construction. Thus a parsing algorithm can be expressed in terms of a deductive reasoner that starts out with a set of premises or axioms then proceeds to recursively apply its rules of inference to the axioms as well as the results of previous applications of the same rules. A deductive reasoner aims to show that such rule application to a set of premises will lead to a certain conclusion. By analogy, a deductive

[10]This rule is itself a conditional where what is above the horizontal line is the antecedent and what is below it the consequent.

Table 3: Converting a MG Parser into a Deductive Reasoning System

MG Parser	$\longmapsto$	Deductive Reasoner
Lexicon	$\longmapsto$	Axioms
$\{Unmerge, Unmove, Scan\}$	$\longmapsto$	Inference Rules
Distinguished Categories	$\longmapsto$	Goal Items

parser would strive to prove that a certain set of axioms will lead a derivation to a certain goal item.

In (6) this goal item is expressed by a fully recognized node "D". Such nodes are referred to as *root nodes* by syntacticians and as *distinguished categories* in some descriptions of MGs (Harkema 2001). They are the same as the "C" category from (5).

Table 3 sums up the conversion of an MG parser into a deductive system. We are finally ready to consider Weinberg's implementation of Minimalism.

4 The Weinberg Parser

Practically a contemporary of the MGs, the Minimalist parser proposed by Amy Weinberg . . . Whereas Stabler's approach something here about the approaches. . . can't think what yet stabler is formal weinberg isnt

4.1 Definition

The definition of the parser is provided below verbatim, because it is very concise. This definition is then discussed in detail with special attention paid to the way it compares with a Stabler-style MG. Table 4 summarizes the comparison of the two parsers.

> A derivation proceeds left to right. At each point in the derivation, merge using the fewest operations needed to check a feature on the category about to be attached. If merger is not possible, try to insert a trace bound to some element within the current command path. If neither merger nor movement is licensed, spell out the command path. Repeat until all terminals are incorporated into the derivation. (*section 11.3, top of p. 290*)

This parser proceeds from left to right through the input string. In this respect it behaves exactly like Stabler's MG parser. The application of structure building operations, despite being described as proceeding from an attempt first

at merger then at movement, functions essentially the same way[11] as in the recognizer defined in **Section 3**.

If neither operation can apply, Weinberg states that the current command path (read: structure built up to this point) is "spelled out". The application of Spellout to an expression takes said expression away from the syntactic component, making it unavailable to the parser. This process plays a pivotal role when it comes to ambiguity. Firstly, it can lead certain input to be attached differently based on what structures are available at the time the input is read in. Secondly, in the case of a parse gone wrong it prevents the parser from reanalyzing the structures it already built so as to correct its mistakes.

Nothing in Stabler's grammar is explicitly analogous to this. One could argue, however, that both (i) feature deletion and (ii) removing structures from the stack once they have been completely recognized yield a similar result.

The way features are checked is slightly but, in my opinion crucially, different than the analogous process in an MG parser. For Weinberg only the category/node being attached or moved is of importance. Nothing is mentioned about the features of the structure already built. In contrast, for Stabler every application of an operation involves checking the features of both of the expressions involved. If anything, the selector/licensor item is more important for Stabler, as it is those features that trigger the application of operations. Is item does not, however, necessarily have to be the item attached or moved, more often it is the opposite.

I believe this difference will become very important when we consider the analyses of preposed objects/matrix subjects.

[11] The order in which the conditions for a certain operation are checked is only relevant if there are cases when the satisfaction of the same conditions leads to the application of different rules. This is not such a case: a feature may trigger either the *Move* and *Merge* operation, but not both.

Table 4: Comparing the Parsers

	Stabler Parser	Weinberg Parser
Left to Right?	Yes	Yes
Order of structure building operations	None. The type of operation depends on the nodes being processed	Try Merge first, then Move
If no operation is applicable?	Reduce/Remove category from stack	Spellout
Extra-grammatical constraints and principles	None.	Economy conditions and the θ-Criterion
Feature-checking	Two-sided: both the selecting/licensing head and the selected/licensed head have their features checked during merger or movement	One-sided: only the node being attached/moved has its features checked

4.2 Economy Conditions

In addition to this the derivation must respect Chomsky's economy conditions (Chomsky 1993) as well as the θ-Criterion. Weinberg cites two assumptions about the application of structure building operations that make a parser minimalist.

(7) *Last Resort*
Operations do not apply unless required to satisfy a constraint. The minimal number of operations is applied to satisfy the constraint.

(8) *Greed (quoting Chomsky 1995)*
"The operation cannot apply to α to enable some different element β to satisfy its [the operation's] properties. . . Benefiting other elements is not allowed."

This is something that may seem to distinguish the Weinberg parser from MGs, where no extra-grammatical concerns or principles are explicitly considered aside from the Shortest Movement Constraint on the Move operation. It can

be argued, however, that at least some of these restrictions are implicitly present in an MG. For instance, an MG operation cannot apply unless it is licensed by the features of the category or categories it is to operate on. Consequently, an operation cannot apply to any other categories and thus principle (8) is satisfied. This is further reinforced by the Shortest Movement Constraint on the *Move* operation that ensures only one licensee-feature-carrying item is selected to move. The first part of principle (7) can be compared to the restriction the category features impose on the kind of operation (*Merge* or *Move*) that can be applied at a given point in the derivation. As for the second part of principle (7), postulating only one operation per feature pair is in fact very close postulating the minimum number of operations for creating trees.

5 Needs new name

This chapter takes a look at how some cases of ambiguity are handled by an MG-based recognizer and continues to (now empirically) compare it to the Weinberg Minimalist parser. In this paper we will focus namely on Argument/Adjunct attachment ambiguities as they are mentioned in 11.4.1 in (Weinberg 1999).

5.1 Preposed Object/Matrix Subject Sentences

Here we will focus on the problem of the ambiguity found in sentences where certain noun phrases can be interpreted as either objects of the preceding verb in a preposed clause (9a) or as subjects of the matrix clause (9b). In the former case humans have much more difficulty parsing the sentence than in the latter.

(9) a. After Mary mended the socks fell off the table
    b. After Mary mended the socks they fell off the table

Weinberg argues that the reason (9a) is so hard to process lies in the requirement that the noun phrase *the socks* have its Case and θ-features be checked. Thus the source of the ambiguity, the first position in the string at which our parser has to choose which clause to attach *the socks* to is that determiner phrase itself.

Weinberg's localization of the choice point between two possible parses seems to ignore that *mends* can actually be both intransitive and transitive. Since both Weinberg and Stabler are assuming a left-to-right parsing sequence, the choice between these two interpretations of *mends* should directly affect whether the parser treats *the socks* as the verb's object or not. This makes it a more appropriate decision point at which ambiguity is introduced. Moreover, assuming a

left-to-right parsing strategy (which both Stabler and Weiberg are), this verb is processed by the parser before *the socks*.

Why is it then that *the socks* and not *mends* is treated as the decision point by the Weinberg parser? It seems this is due to what constituent the parser focuses on for feature checking.

5.2 The Role of Feature Checking

As was mentioned above, the Weinberg parser sees the constituent being attached as the only point at which ambiguity can be introduced. This is because its features are the ones that need to be checked (in the case of *the socks* it has to be assigned a theta role). The verb *mends* in (9a) cannot be the source of ambiguity under this interpretation because at the time of its attachment, when its features are in focus, it does not have them checked but instead (in one of its meanings) introduces argument features that will need to be received later in the parse by any available for that DPs. These features are not relevant to the Weinberg parser until their matches are introduced by *the socks*. By then, however, the features in focus are those of the DP. Thus the decision of which clause to attach *the socks* becomes the source of ambiguity in a Weinberg parser only when that DP itself is being processed.

Such a placement of the decision point seems intuitively somewhat inaccurate. In **Section 6** it will be demonstrated that a Stabler parser presents a more plausible analysis of this construction. This analysis will also highlight an aspect of parsing that is not adressed in (Weinberg 1999): the question of how lexical ambiguity is resolved and how this affects sentence processing.

For now, however, we will consider some other cases presented by Weinberg, paying close attention to how ambiguity is introduced and handled.

5.3 Direct Object/Complement Subject Ambiguity

Weinberg's analysis of the sentences in (10) and (11) is very much the same as of those in (9). In this case, again, the ambiguity, per Weinberg, shows its head only when "his sister" is processed, but not before.

(10) the man believed his sister to be a genius

(11) the man believed his sister

Not surprisingly, the same concerns about the analysis as expressed in the previous section apply to these sentences. These examples also further demonstrate the relationship between MG-style feature checking and θ-role assignment as presented by Weinberg. As before, they show that the former can account for the intuitions behind stipulating the latter without explicitly referring to it.

5.4 Ditransitive/Complex Transitive Object Ambiguity

What is interesting about examples (12) and (13) is that an MG parser does not need to assume a Larsonian VP structure in order to account for the fact that (12) is preferred. Simply assuming a ditransitive feature set for "gave", not an unreasonable assumption, achieves the same result. Once again θ-role assignment is not necessary to explain the data.

(12) John gave the man the dog for Christmas

(13) John gave the man the dog bit a bandage

5.5 Subcategorized PP/NP Modifier Ambiguity

6 Determining the Source of Ambiguity

6.1 Lexical Ambiguity Resolution

Lexical Ambiguity in parsing occurs when the recognizer processes a string that has more that one possible feature set associated with it. Consider the sentences in (14).

(14) a. Abu never eats meat.

b. Abu never eats.

The only difference between them is that in sentence (14a) *eats* takes a direct object, whereas in (14b) it does not. In a Stabler-based grammar's Lexicon, these two versions of *eats* would have two corresponding feature sets (shown in (15)) associated with it. These versions are shown in , where (15a) corresponds to (14a) and (15b) to (14b).

(15) Two different feature sets (lexical entries) for *eats*

a. =d, v

b. v

It is of course trivial to establish which feature set is the correct one for a particular parse given the full sentence, but a left-to-right parser does not have that luxury. At the point when it parses *eats* the information it has access to looks like (14b), but has the potential of becoming like (14a) if the next word processed by the parser happens to be a phrase with feature d.

One can easily imagine this as a problem for a parsing algorithm. How is it to determine in case of multiple lexical entries for a word which one of these entries will ultimately lead to the correct parse? Does it have to determine this at all, or is it possible to entertain all or several different variants of a derivation? There is

some research (Bever 1970) suggesting that humans tend to do the former: they choose one of the possible parses and then follow through with it as far as they can.

In the best case scenario, the parse is correct and the sentence is successfully processed. In the worst case, the derivation gets "garden-pathed" and the sentence is not processed successfully. This trait can be formally expressed by introducing a component in the Grammar responsible for resolving any ambiguities as they arise in the parsing process.

We will adopt the term ORACLE for this component, a word used in the literature to describe loosely similar concepts[12]. While neither Stabler's nor Weinberg's parsers in their original definitions explicitly tackle this problem, neither proposal is incompatible with such an ORACLE component. For the purposes of this paper the precise functioning of the ORACLE is irrelevant, so we may treat it as a black-box kind of mechanism that consistently favors certain lexical entries over others.

It will also be taken for granted that the ORACLE interacts the same way with both parsing approaches under consideration. Put differently, both in the Stabler and the Weinberg parsers the ORACLE's role is to somehow pick one lexical entry to be processed whenever an input token warrants multiple possible candidates for that role. With this in mind, the next section will revisit the data in (9) and observe how the introduction of an ORACLE affects the explanations of the source of ambiguity in that sentence.

6.2 Falling Socks Revisited

The ORACLE will function the same way with both parsers, hence the differences between them can be ignored temporarily. Thus 'Parser' for the sake of the following derivation example will refer equally to the Stabler recognizer as well as the Weinberg model.

A derivation of (9) would proceed this way. *After* and *Mary* would get processed the same way the Parser would have parsed them without the ORACLE. By contrast, *mended* would have to be passed to the ORACLE before the Parser could start working with it. The ORACLE would determine whether to select the transitive or the intransitive meaning of *mended* (the feature bundles in (15a) and (15b) respectively). This decision, **crucially**, is not influenced in any way by what (if any) input still remains to be processed.

Weinberg's requirement of θ-feature checking on *the socks* fails to thoroughly explain the data, because of the irrelevance the following input. This can be

[12]John Hale uses *oracle* when refering to the mechanism in his *rational parser* that guides it to the optimal derivation (Hale 2011).

seen clearly if we entertain the apparently unlikely scenario in which the ORACLE chooses the intransitive meaning of *mended*. As far as the ORACLE is concerned this decision is as valid as its opposite. However, no amount of strengthening of the θ-Criterion can negate the fact that by the time *the socks* is processed, *mended*, being intransitive, will simply not have the right θ role to assign.

The advantage of the Stabler recognizer in this scenario is that it lacks the extra-grammatical stipulations about θ-roles included in the Weinberg parser. In a Stabler parser, if the ORACLE happens to choose the lexical entry for *mended* that does not require a direct object (and as a consequence does not assign the relevant θ role), *the socks* is automatically considered for the subject role in the matrix clause. In effect, the resulting derivation is identical to the one for the unambiguous sentence (16).

(16) After Mary yawned the socks fell off the table

This reanalysis of (9) shows that the extra-grammatical stipulations present in Weinberg are not needed to account for the sentences featuring preposed Object/Matrix Subject ambiguity. Moreover, it can certainly be argued that in this particular case a Stabler-style explanation of the ambiguity as well as the localization of its source is more in line with the Minimalist Program, because such an explanation does not require extragrammatical principles such as the θ-Criterion.

7 Conclusions and Further Work

This short excursion into formal Minimalist Grammars has brought to light several interesting points about parser design. First of all, it was established that by following Stabler in shifting the focus to mutual feature checking[13] we were able to explain some of the data thought to have been accounted for by extra-grammatical priciples like the θ-Criterion. Secondly, we provided a small demonstration of how models of the same process that were originally developed in different fields with slightly differing implementations can be compared. This is important because interdisciplinary approaches to problems often prove productive and it is consequently important to be able to syntesize results and theories from different fields that all share a common topic.

That being said there is still much room for further research in this area and some questions and issues touched upon in this paper are far from settled. For example, the exact workings of the ORACLE most definitely warrant attention. The importance of ORACLE's decision-making capabilities was clearly shown in this

[13] Mutual refers to both the items triggering the application of an operation well as items being selected by the operation having their features checked.

paper. It is thus only natural to ask how exactly this decision-making functions exactly.

There are already some attempts made at solving the problem of deciding which derivational path to follow (Hale 2011; Mainguy 2010). It would be interesting to survey these results in more detail and perhaps discover some generalizations or new questions to ask.

The role of Lexical Ambiguity in parsing is another topic to be explored. One strong claim that could be made is that in Minimalist Grammars lexical ambiguity is the only source of ambiguity, since the Lexicon is the only part of the input into the system that varies across grammars, whereas the features and especially the operations remain static. While this claim may end up being ultimately being too strong to be defendable, an attempt at studying it would definitely help clarify the role that Lexical Ambiguity plays in human sentence processing.

8 Acknowledgements

This paper is the result of ongoing work on exploring properties of minimalist parsers as part of an Independent Study at UMass Amherst. My endless gratitude goes to professor Brian Dillon for his invaluable guidance, infectuous enthusiasm and shrewd insights and criticisms. I would like to mention Felix Lehmann for sharing his LaTeX expertise and providing extremely helpful advice on formatting. Finally, I extend my deepest thanks to Amanda Rysling and my mother Tatiana P. Kurenkov. Without their comments and suggestions this paper in its current state would not have been possible.

References

Bever, Thomas G (1970). "The cognitive basis for linguistic structures". In: *Cognition and the development of language* 279.362, pp. 1–61.

Chomsky, Noam (1993). "A minimalist program for linguistic theory". In: *The view from Building 20: essays in linguistics in honor of Sylvain Bromberger*. Ed. by Kenneth Hale and Samuel Keyser. Studies in Linguistics 24. Cambridge, MA: MIT Press, pp. 1–52.

— (1995). *The minimalist program*. Studies in Linguistics 28. Cambridge, Massachusetts: MIT Press.

Demers, Alan J (1977). "Generalized left corner parsing". In: *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*. ACM, pp. 170–182.

Frey, Werner and Hans-Martin Gärtner (2002). "On the treatment of scrambling and adjunction in minimalist grammars". In: *Proceedings of Formal Grammar*. Vol. 2002, pp. 41–52.

Hale, John T (2011). "What a rational parser would do". In: *Cognitive Science* 35.3, pp. 399–443.

Harkema, Hendrik (2001). "Parsing minimalist languages". PhD thesis. Citeseer.

Johnson-Laird, Philip N (1983). *Mental models: Towards a cognitive science of language, inference, and consciousness*. 6. Harvard University Press.

Mainguy, Thomas (2010). "A probabilistic top-down parser for minimalist grammars". In: *arXiv preprint arXiv:1010.1826*.

Michaelis, Jens (2001). "Derivational minimalism is mildly context–sensitive". In: *Logical aspects of computational linguistics*. Springer, pp. 179–198.

Resnik, Philip (1992). "Left-corner parsing and psychological plausibility". In: *Proceedings of the 14th conference on Computational linguistics-Volume 1*. Association for Computational Linguistics, pp. 191–197.

Shieber, Stuart M, Yves Schabes, and Fernando CN Pereira (1995). "Principles and implementation of deductive parsing". In: *The Journal of Logic Programming* 24.1, pp. 3–36.

Stabler, Edward (1997a). "Derivational Minimalism". In: *Logical Aspects of Computational Linguistics*. Ed. by Christian Retoré. Lecture Notes in Artificial Intelligence 1328. Heidelberg: Springer Verlag, pp. 68–95.

— (1997b). "Derivational minimalism". In: *Logical aspects of computational linguistics*. Springer Berlin Heidelberg, pp. 68–95.

Weinberg, Amy (1999). "A minimalist theory of human sentence processing". In: *Working minimalism*. Ed. by Samuel Epstein and Norbert Hornstein. Cambridge, Massachusetts: MIT Press, pp. 283–315.

A Appendix: Intransitives

a word about unergatives and unaccusatives

B Appendix: Parser Documentation

As part of working out a Minimalist Parser model a program was created in Python. It follows the specifications outlined in **Section 3**. Stabler's formalism was chosen over Weinberg's as a basis for a program due to its formal clarity and consequently the ease with which it could be transferred into code.

B.1 `recognizer`

demo-mode

core functions.

plotting