# A Sound and Complete Left-Corner Parsing for Minimalist Grammars

**Miloš Stanojević**
University of Edinburgh
Edinburgh EH8 9AB, UK
m.stanojevic@ed.ac.uk

**Edward P. Stabler**
UCLA and Nuance Communications
California, USA
edward.stabler@nuance.com

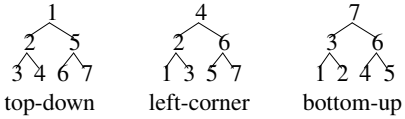
## Abstract

This paper presents a left-corner parser for minimalist grammars. The relation between the parser and the grammar is transparent in the sense that there is a very simple 1-1 correspondence between derivations and parses. Like left-corner context-free parsers, left-corner minimalist parsers can be non-terminating when the grammar has empty left corners, so an easily computed left-corner oracle is defined to restrict the search.

## 1 Introduction

Minimalist grammars (MGs) (Stabler, 1997) were inspired by proposals in Chomskian syntax (Chomsky, 1995). MGs are strictly more expressive than context free grammars (CFGs) and weakly equivalent to multiple context free grammars (MCFGs) (Michaelis, 2001; Harkema, 2001a). The literature presents bottom-up and top-down parsers for MGs (Harkema, 2001b), which differ in the order in which derivations are constructed, and consequently they may differ in their memory demands at each point in the parse. But partly because of those memory demands, parsers that mix top-down and bottom-up steps are often regarded as psycholinguistically more plausible (Hale, 2014; Resnik, 1992; Abney and Johnson, 1991).

Among mixed strategies, left-corner parsing (LC) is perhaps the best known (Rosenkrantz and Lewis, 1970). A left-corner parser does not begin by guessing what's in the string, as a top-down parser does. But it also does not just reduce elements of the input, as a bottom-up parser does. A left-corner parser looks first at what is in the string (completing the left-most constituent, bottom-up) and then predicting the sisters of that element (top-down), if any. The following CFG trees have nodes numbered in the order they would be constructed by bottom-up, left-corner and top-down strategies:




LC parsing is bottom-up on the leftmost leaf, but then proposes a completed parent of that node on condition that its predicted sister is found.

For CFGs, LC parsing is well understood (Aho and Ullman, 1972; Rosenkrantz and Lewis, 1970). In a CF rule $A \rightarrow B\ C$, the left corner is of course always $B$. Johnson and Roark (2000) generalize from CFGs to unification-based grammars and show how to allow some selected categories to be parsed left-corner while others are parsed top-down. Extending these ideas to MGs, we must deal with movements, with rules that sometimes have their first daughter on the left and sometimes on the right, and with categories that are sometimes empty and sometimes not. Left corner parsers were developed for some other discontinuous formalisms with similar properties (van Noord, 1991; Díaz et al., 2002) but in all cases these parsers fall in the category of the arc-standard left corner parsing. Here we present a left corner parser that is of arc-eager type which is argued to be more cognitively plausible due to its higher degree of incrementality (Abney and Johnson, 1991; Resnik, 1992).

A first approach to left-corner MG parsing, designed to involve a kind of psycholinguistically motivated search, has been presented (Hunter, 2017), but that proposal does not handle all MGs. In particular, remnant movement presents the main challenge to Hunter's parser. The parser proposed here handles all MGs, and it is easily shown to be sound and complete via a simple 1-1 correspon-



*Proceedings of the Eighth Workshop on Cognitive Aspects of Computational Language Learning and Processing*, pages 65–74
Melbourne, Australia, July 19, 2018. 

dence between derivations and parses. (However, as mentioned in the conclusion, the present proposal does not yet address the psycholinguistic issues raised by Hunter.) Following similar work on CFGs (Pereira and Shieber, 1987, §6.3.1), we show how to compute a left-corner oracle that can improve efficiency. And probabilities can be used in a LC beam-parser to pursue the most probable parses at each step (Manning and Carpenter, 1997).

## 2 Minimalist grammars

We present a succinct definition adapted from Stabler (2011, §A.1) and then consider a simple example derivation in Figure 1. An MG G=$\langle \Sigma, B, Lex, C, \{\text{merge,move}\}\rangle$, where $\Sigma$ is the **vocabulary**, $B$ is a set of **basic features**, $Lex$ is a finite **lexicon** (as defined just below), $C \in B$ is the **start category**, and {merge,move} are the generating functions. The basic features of the set $B$ are concatenated with prefix operators to specify their roles, as follows:

**categories, selectees** $= B$
**selectors** $= \{\text{=f} \mid \text{f} \in B\}$
**licensees** $= \{\text{-f} \mid \text{f} \in B\}$
**licensors** $= \{\text{+f} \mid \text{f} \in B\}$.

Let $F$ be the set of role-marked **features**, that is, the union of the categories, selectors, licensors and licensees. Let $T$={`::`, `:`} be two **types**, indicating 'lexical' and 'derived' structures, respectively. Let $\mathbb{C} = \Sigma^* \times T \times F^*$ be the set of **chains**. Let $E = \mathbb{C}^+$ be the set of **expressions**. An expression is a chain together with its 'moving' sub-chains, if any. Then the **lexicon** $Lex \subset \Sigma^* \times \{::\} \times F^*$ is a finite set. We write $\epsilon$ for the empty string. Merge and move are defined in Table 1. Note that each merge rule deletes a selection feature =f and a corresponding category feature f, so the result on the left side of the rule has 2 features less than the total number of features on the right. Similarly, each move rule deletes a licensor feature +f and a licensee feature -f. Note also that the rules have pairwise disjoint domains; that is, an instance of a right side of a rule is not an instance of the right side of any other rule. The set of **structures**, everything you can derive from the lexicon using the rules, S(G)=closure(Lex,{merge,move}). The **sentences L(G)** $= \{s \mid s \cdot C \in \text{S(G)}$ for some type $\cdot \in \{:, ::\}\}$, where $C$ is the 'start' category.

Example grammar **G1** with start category c uses features +wh and -wh to trigger wh-movements:

| | |
|---|---|
| $\epsilon$ :: =v c | knows :: =c =d v |
| $\epsilon$ :: =v +wh c | likes :: =d =d v |
| Aca :: d | what :: d -wh |
| Bibi :: d | |

These 7 lexical items define an infinite language. An example derivation is shown in Figure 1.

Grammar G1 is simple in a way that can be misleading, since the mechanisms that allow simple wh-movement also allow remnant movements, that is, movements of a constituent out of which something has already moved. Without remnant movements, MGs only define context-free languages (Kobele, 2010). So remnant movements are responsible for deriving copying and other sorts of crossing dependencies that cannot be enforced in a CFG. Consider **G2**:

| | |
|---|---|
| $\bot$ :: T -r -l | $\top$ :: =T +r +l T |
| a :: =A +l T -l | a :: =T +r A -r |
| b :: =B +l T -l | b :: =T +r B -r |

With T as the start category, this grammar defines the copy language $\bot$XX$\top$ where X is any string of *a*'s and *b*'s. Bracketing the reduplicated string with $\bot$ and $\top$ allows this very simple grammar with no empty categories, and makes it easy to track how the positions of these elements is defined by the derivation tree on the left in Figure 2, with 6 movements numbered 0 to 4, with TP(0) moving twice.

This example shows that simple mechanisms and simple lexical features can produce surprising patterns. Some copy-like patterns are fairly easy to see in human languages (Bresnan et al., 1982; Shieber, 1985), and many proposals with remnant derivations have become quite prominent in syntactic theory, even where copy-like patterns are not immediately obvious (den Besten and Webelhuth, 1990; Kayne, 1994; Koopman and Szabolcsi, 2000; Hinterhölzl, 2006; Grewendorf, 2015; Thoms and Walkden, 2018). Since remnant-movement analyses seem appropriate for some constructions in human languages, and since grammars defining those analyses are often quite simple, and since at least in many cases, remnant analyses are easy to compute, it would be a mistake to dismiss these derivations too quickly. For present purposes, the relevant and obvious point is that a sound and complete left corner parser for MGs must handle all such derivations.

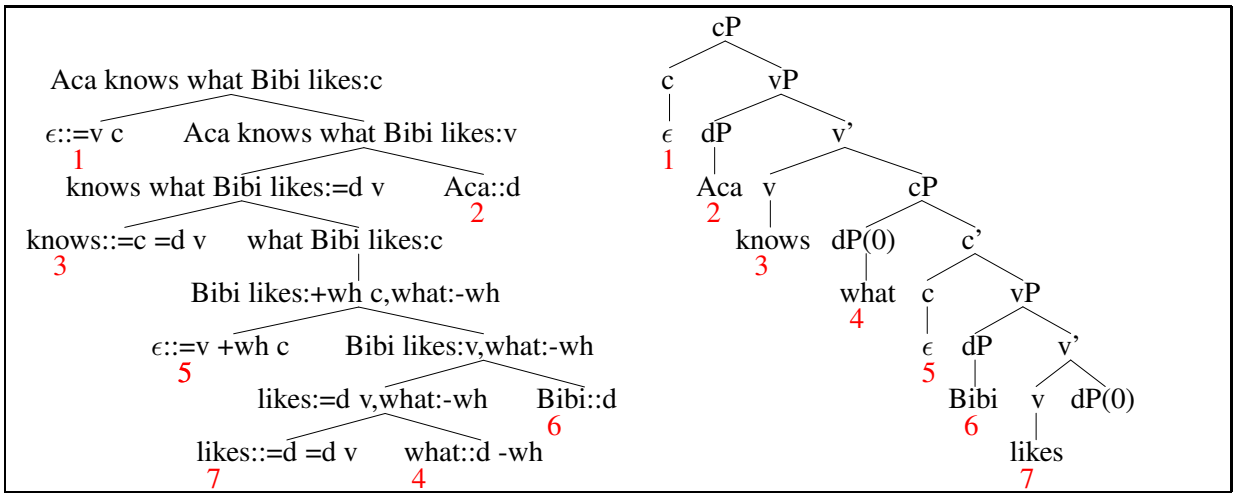



Figure 1: **Derivation tree** from **G1** on the left, and corresponding X-bar **derived tree** on the right. In the derivation tree, the binary internal nodes are applications of merge rules, while the unary node is an application of move1. Computing the derived X-bar structure from the derivation is briefly described in §5 below. Note that in the X-bar tree, P is added to each category feature when the complex is the 'maximal projection' of the head, while primes indicate intermediate projections, and the moved constituent is 'coindexed' with its origin by marking both positions with (0). For the LC parser, the derivation tree (not the derived X-bar tree) is the important object, since the derivation is what shows whether a string is derived by the grammar. But which daughter is 'leftmost' in the derivation tree is determined by the derived string positions, counted here from 1 to 7, left to right. Derived categories become left corners when they are completed, so for the nodes in the derivation tree, the leftmost daughter, in the sense relevant for LC parsing, is the one that is completed first in the left-to-right parse of the derived string.

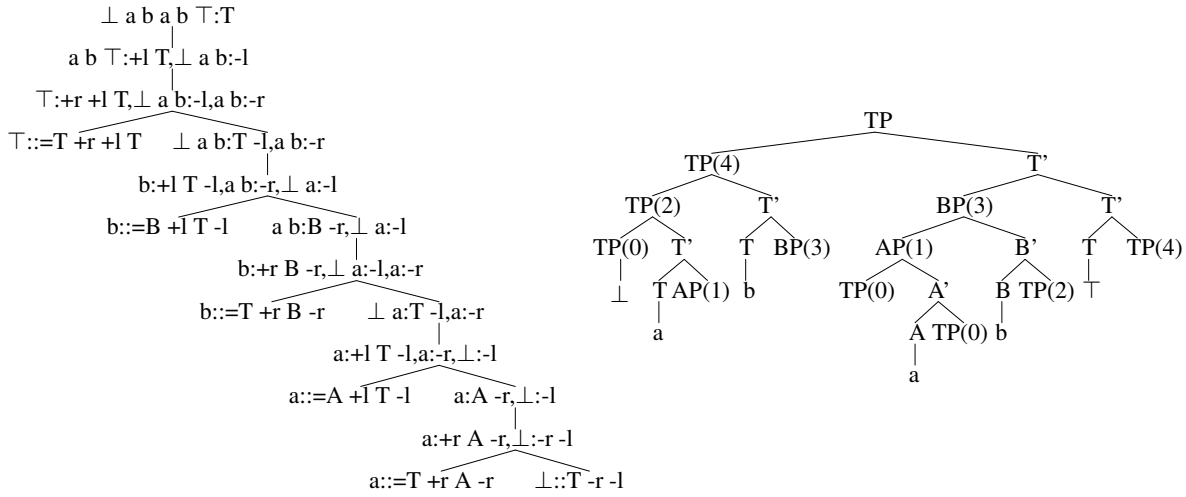



Figure 2: **Derivation tree** from **G2** on the left, and corresponding **derived tree** on the right. Note that the empty TP(0) moves twice, first with MOVE2 and then landing with MOVE1. That TP is just the empty head, the only element of G2 with 2 licensees. Graf et al. (2016) show that all MG languages can be defined without moving any phrase more than once, but G2 is beautifully small and symmetric.

**merge** is the union of the following 3 rules, each with 2 elements on the right,
for strings $s, t \in \Sigma^*$, for types $\cdot \in \{:, ::\}$ (lexical and derived, respectively),
for feature sequences $\gamma \in F^*$, $\delta \in F^+$, and for chains $\alpha_1, \ldots, \alpha_k, \iota_1, \ldots, \iota_l$ $(0 \leq k, l)$

(MERGE1) lexical item $s$ selects non-mover $t$ to produce the merged $st$

$$st : \gamma, \alpha_1, \ldots, \alpha_k \quad \leftarrow \quad s :: =f\gamma \qquad t \cdot f, \alpha_1, \ldots, \alpha_k$$

(MERGE2) derived item $s$ selects a non-mover $t$ to produce the merged $ts$

$$ts : \gamma, \alpha_1, \ldots, \alpha_k, \iota_1, \ldots, \iota_l \quad \leftarrow \quad s : =f\gamma, \alpha_1, \ldots, \alpha_k \qquad t \cdot f, \iota_1, \ldots, \iota_l$$

(MERGE3) any item $s$ selects a mover $t$ to produce the merged $s$ with chain $t$

$$s : \gamma, \alpha_1, \ldots, \alpha_k, t : \delta, \iota_1, \ldots, \iota_l \quad \leftarrow \quad s \cdot =f\gamma, \alpha_1, \ldots, \alpha_k \qquad t \cdot f\delta, \iota_1, \ldots, \iota_l$$

**move** is the union of the following 2 rules, each with 1 element on the right,
for $\delta \in F^+$, such that none of the chains $\alpha_1, \ldots, \alpha_{i-1}, \alpha_{i+1}, \ldots, \alpha_k$ has $\text{-}f$ as its first feature:

(MOVE1) final move of $t$, so its $\text{-}f$ chain is eliminated on the left

$$ts : \gamma, \alpha_1, \ldots, \alpha_{i-1}, \alpha_{i+1}, \ldots, \alpha_k \quad \leftarrow \quad s : +f\gamma, \alpha_1, \ldots, \alpha_{i-1}, t : \text{-}f, \alpha_{i+1}, \ldots, \alpha_k$$

(MOVE2) nonfinal move of $t$, so its chain continues with features $\delta$

$$s : \gamma, \alpha_1, \ldots, \alpha_{i-1}, t : \delta, \alpha_{i+1}, \ldots, \alpha_k \quad \leftarrow \quad s : +f\gamma, \alpha_1, \ldots, \alpha_{i-1}, t : \text{-}f\delta, \alpha_{i+1}, \ldots, \alpha_k$$

Table 1: **Rules for minimalist grammars** from (Stabler, 2011, §A.1). Where a CFG has $\rightarrow$, these rules have $\leftarrow$ as a reminder that they are usually used 'bottom-up', as functions from the elements on their right sides to the corresponding value on the left. To handle movements, MGs show the strings $s, t$ explicitly. And where CFG rules have categories, these rules have complexes, i.e. comma-separated chains. Intuitively, each chain is a string with a type and syntactic features, and each constituent on either side of these rules is a sequence of chains, an initial head chain possibly followed by moving chains.

## 3 Left corner MG parsing

A left corner parser uses an MG rule when the leftmost element on the right side is complete, where by leftmost element we do not mean the one that appears first in the rules of Table 1. Rather, the leftmost element is the one that is completed first in the left-to-right parse. For MOVE rules, there is just one element on the right side, so that element is the left-corner. When the right side of a MOVE rule is complete, it is replaced by the corresponding left side. But matters are more interesting for MERGE rules, which have two constituents on their right sides. Because the first argument $s$ of MERGE1 is lexical, it is always the left corner of that rule. But for MERGE2 and MERGE3, either argument can have moved elements that appear to the right, so which argument is the left corner depends on the particular grammar and even sometimes on the particular derivation.

In the derivation shown in Figure 1, for example, there is one application of MERGE3, to combine *likes* with *what*, and in that case, the selectee lexical item *what* is the left corner because it is the 4th terminal element, while its sister in the derivation tree is terminal element 7. In Figure 2, we can see that $\perp$ occurs first in the input, and is processed in the very first step of the successful left corner parse, even though it is the deepest, rightmost element in the derivation tree.

The MERGE3 rule of MGs raises another tricky issue. After the output of this rule with the predicted right corner is computed, we need to remember it, sometimes for a number of steps, since left and right corners can be arbitrarily far apart. Even with the simple G1, we can get *Aca knows what Bibi knows Aca knows Bibi knows…Aca likes*. We could put the MERGE3 output into a special store, like the HOLD register of ATNs (Wanner and Maratsos, 1978), but here we adopt the equivalent strategy of keeping MERGE3 predictions in the memory that holds our other completed left corners and predicted elements. We call this memory a queue, since it is ordered like a stack, but the parser can access elements that are not on top, as explained below. Queue could be treated as a multiset (since elements can be accessed even if they are not on the top) but treating queue as an ordered structure allows easier defini-

tion of oracle and easier definition of which constituent is triggering the next parser's operation.

It will be convenient to number string positions as usual: *0 Aca 1 knows 2 what 3 Bibi 4 likes 5*. Substrings can then be given by their spans, so *Aca* in our example is represented by 0-1, *knows* is 1-2, and an initial empty element would have the span 0-0.

So the parser state is given by

(remaining input, current position, queue),

and we begin with

(input, 0, $\epsilon$).

For any input of length $n$, we then attempt to apply the LC rules to get

($\epsilon$, $n$, 0-$n$·c),

where · is any type and c is the start category. The LC rules are these:

**(0)** The SHIFT rule takes an initial (possibly empty) element $w$ with span x-y from the beginning of the remaining input, where the lexicon has $w :: \gamma$, and puts x-y::$\gamma$ onto the queue.

**(1)** For an MG rule R of the form $A \leftarrow B\ C$ with left corner $B$, if an instance of $B$ is on top of the queue, lc1(R) removes B from the top of the queue and replaces it with an element $C \Rightarrow A$. Since any merge rule can have the selector as its left corner, we have the LC rules LC1(MERGE1), LC1(MERGE2), and LC1(MERGE3).

Let's be more precise about being 'an instance'. When R is $A \leftarrow B\ C$, the top element $B'$ of the queue is **an instance of** B iff we can find a (most general) substitution $\theta$ such that $B'\theta = B\theta$. In that case, lc(R) replaces $B'$ with $(C \Rightarrow A)\theta$. This computation of substitutions can be done by standard unification (Lloyd, 1987). For example, looking at MERGE1 in Table 1, note that the first constituent on the right specifies the feature $f$, the sequence $\gamma$, and the string $s$, but not the string $t$ or the 0 or more moving chains $\alpha_1, \ldots, \alpha_k$. So when LC1(MERGE1) applies, the unspecified elements are left as variables, to be instantiated by later steps. So when $s :: {=}f\gamma$ (for some particular $s, f, \gamma$) is on top of the queue, LC1(MERGE1) replaces it by

$$(\underline{t} : f, \underline{\alpha_1, \ldots, \alpha_k} \Rightarrow s\underline{t} : \gamma, \underline{\alpha_1, \ldots, \alpha_k}).$$

where underlined elements are variables.

**(2)** For an MG rule R of the form $A \leftarrow B\ C'$ with completed left corner $C$ and $C\theta = C'\theta$, lc2(R) replaces $C$ on top of the queue by $(B \Rightarrow A)\theta$. For this case, where the second argument on the right side is the left corner, we have the LC rules LC2(MERGE2) and LC2(MERGE3).

**(3)** Similarly for MG rules $A \leftarrow B$, the only possible leftcorner is a constituent $B$ where $B\theta = B'\theta$, replacing $B'$ by $A\theta$. So we have LC1(MOVE1) and LC1(MOVE2) in this case.

**(4)** We have introduced 8 LC rules so far. There is SHIFT, and there are 7 LC rules corresponding to the 5 MG rules in Table 1, because of the fact that the left corner of MERGE2 and MERGE3 can be either the first or second element on the right side of the rule. Each LC rule acts to put something new on top of the queue. The 'arc-eager' variant of LC parsing, which we will define here, adds additional variants of those 8 rules: instead of just putting the new element on top of the queue, the element created by a rule can also be used to complete a prediction on the queue, 'connecting' the new element with structure already built.[1] Importantly, the following completion variants of the LC rules can search below the top element to find connecting elements:

**c(R)** If LC rule R creates a constituent $B$, and the queue has $B' \Rightarrow A$, where $B\theta = B'\theta$, then c(R) removes $B' \Rightarrow A$ puts $A\theta$ onto the queue.

**c1(R)** If LC rule R creates $B \Rightarrow A$ and we already have $C \Rightarrow B'$ on the queue, where $B\theta = B'\theta$, then c1(R) removes $C \Rightarrow B'$ and puts $(C \Rightarrow A)\theta$ onto the queue.

**c2(R)** If LC rule R creates $C \Rightarrow B$ and we already have $B' \Rightarrow A$ on the queue, where $B\theta = B'\theta$, c2(R) removes $B' \Rightarrow A$ and puts $(C \Rightarrow A)\theta$ onto the queue.

**c3(R)** If LC rule R creates a constituent $C \Rightarrow B$ and we already have $B' \Rightarrow A$ and $D \Rightarrow C'$ on the queue, where $B\theta = B'\theta$ and $C\theta = C'\theta$ c3(R) removes $B' \Rightarrow A$ and $D \Rightarrow C'$ and puts $(D \Rightarrow A)\theta$ onto the queue.

These completion rules are similar to the 'composition' rules of combinatory categorial grammar (Steedman, 2014).

[1] Instead of requiring completions to happen when an element is added to the queue, the 'arc-standard' variant of LC parsing uses separate complete rules, which means that a constituent need not (and sometimes cannot) be connected to predicted structure at the time when it is first proposed.

That completes the specification of an arc-eager left corner parser for MGs. The rules are non-deterministic; that is, at many points in a parse, various different LC rules can apply. But for each $n$-node derivation tree, there is a unique sequence of $n$ LC rule applications that accepts the derived string. This 1-1 correspondence between derivations and parses is unsurprising given the definition of LC. Intuitively, every LC rule is an MG rule, except that it's triggered by its left corner, and it can 'complete' already predicted constituents. This makes it relatively easy to establish the correctness of the parsing method (§5, below).

The 14 node derivation tree in Figure 1 has this 14 step LC parse, indicating the rule used, the remaining input, and queue contents from top to bottom, with variables _M and _N for chain sequences, _Fs for features, _ for span positions, and [] represents the remaining input $\epsilon$ in the last 2 steps of the listing:

```
1. shift [Aca,knows,what,Bibi,likes]
    0-0::=v c
2. lc1(merge1) [Aca,knows,what,Bibi,likes]
    (0-_.v _M  =>  0-_:c _M)
3. shift [knows,what,Bibi,likes]
    0-1::d
    (0-_.v _M  =>  0-_:c _M)
4. c1(lc2(merge2)) [knows,what,Bibi,likes]
    (1-_:=d v _M  =>  0-_:c _M)
5. shift [what,Bibi,likes]
    1-2::=c =d v
    (1-_:=d v _M  =>  0-_:c _M)
6. c1(lc1(merge1)) [what,Bibi,likes]
    (2-_.c _M  =>  0-_:c _M)
7. shift [Bibi,likes]
    2-3::d -wh
    (2-_.c _M  =>  0-_:c _M)
8. lc2(merge3) [Bibi,likes]
    (_-_.=d _Fs_M  =>  _-_:_Fs,2-3:-wh )
    (2-_.c _M  =>  0-_:c _M)
9. shift [Bibi,likes]
    3-3::=v +wh c
    (_-_.=d _Fs  =>  _-_:_Fs,2-3:-wh )
    (2-_.c _M  =>  0-_:c _M)
10. lc1(merge1) [Bibi,likes]
    (3-_.v _M  =>  3-_:+wh c _M)
    (_-_.=d _Fs  =>  _-_:_Fs,2-3:-wh )
    (2-_.c _N  =>  0-_:c _N)
11. shift [likes]
    3-4::d
    (3-_.v _M  =>  3-_:+wh c _M)
    (_-_.=d _Fs  =>  _-_:_Fs,2-3:-wh )
    (2-_.c _N  =>  0-_:c _N)
12. c3(lc2(merge2)) [likes]
    (4-_.=d =d v  =>  3-_:+wh c ,2-3:-wh )
    (2-_.c _M  =>  0-_:c _M)
13. c(shift) []
    3-5:+wh c ,2-3:-wh
    (2-_.c _M  =>  0-_:c _M)
14. c(lc1(move1)) []
    0-5:c
```

The derivation tree in Figure 2 has 17 nodes, and so there is a corresponding 17 step LC parse. For lack of space, we do not present that parse here. It is easy to calculate by hand (especially if you cheat by looking at the tree in Figure 2), but much easier to calculate using an implementation of the parsing method.[2]

## 4 A left corner oracle

The description of the parsing method above specifies the steps that can be taken, but does not specify which step to take in situations where more than one is possible. As in the case of CFG parsing methods, we could take some sequence of steps arbitrarily and then backtrack, if necessary, to explore other options, but this is not efficient, in general (Aho and Ullman, 1972). A better alternative is to use 'memoization', 'tabling' – that is, keep computed results in an indexed chart or table so that they do not need to be recomputed – compare (Kanazawa, 2008; Swift and Warren, 2012). Another strategy is to compute a beam of most probable alternatives (Manning and Carpenter, 1997). But here, we will show how to define an oracle which can tell us that certain steps cannot possibly lead to completed derivations, following similar work on CFGs (Pereira and Shieber, 1987, §6.3.1). This oracle can be used with memoizing or beam strategies, but as in prior work on CFG parsing, we find that sometimes an easily computed oracle makes even backtracking search efficient. Here we define a simple oracle that suffices for G1 and G2. For each grammar, we can efficiently compute a *link* relation that we use in this way: A new constituent $A'$ or $B' \Rightarrow A'$ can be put onto the queue only if $A'$ stands in the LINK relation to a *predicted category*, that is, where the start category is predicted when the queue is empty, and a category $B$ is predicted when we have $B \Rightarrow A$ on top of the queue. For many grammars, this use of a LINK oracle eliminates many blind alleys, sometimes infinite ones.

Let $\text{LINK}(X, Y)$ hold iff at least one of these conditions holds: (1) $X$ is a left corner of $Y$, (2) $Y$ contains a initial licensee -f and the first feature of $Y$ is +f, or (3) $X$ and $Y$ are in the transitive closure of the relation defined by (1) and (2). To keep things finite and simple, the elements related by LINK are like queue elements except the mover

[2] An implementation of this parser and our example grammars is provided at https://github.com/stanojevic/Left-Corner-MG-parser

lists are always variables, and spans are always unspecified. Clearly, for any grammar, this LINK relation is easy to compute. Possible head feature sequences are non-empty suffixes of lexical features, suffixes that do not begin with -f. The possible left corners of those head sequences are computable from the 7 left corner rules above. This simple LINK relation is our oracle.

## 5 Correctness, and explicit trees

We sketch the basic ideas needed to demonstrate the soundness of our parsing method (every successful parse is of a grammatical string) and its completeness (every grammatical string has a successful parse). Notice that while the top-down MG parser in Stabler (2013) needed indices to keep track of relative linear positions of predicted constituents, no such thing is needed in the LC parser. This is because in LC parsing, every rule has a bottom-up left corner, and in all cases except for MERGE3, that left corner determines the linear order of any predicted sisters.

For MERGE3, neither element on the right side of the rule, neither the selector nor the selectee, determines the relative position of the other. But the MERGE3 **selectee** has a feature sequence of the form: f$\gamma$-g, and this tells us that the linear position of this element will be to the left of the corresponding +g constituent that is the left corner of move1. That is where the string part of the -g constituent 'lands'. The Shortest Move Constraint (SMC) guarantees that this pairing of the +g and -g constituents is unique in any well formed derivation, and the well-formedness of the derivation is guaranteed by requiring that constituents built by the derivation are connected by instances of the 5 MG rules in Table 1.

Locating the relevant +g move1 constituent also sufficiently locates the MERGE3 **selector** with its feature sequence of the form =f$\gamma$. It can come from anywhere in the +g move1 constituent's derivation that is compatible with its features. Consequently, when predicting this element, the prediction is put onto the queue when the +g constituent is built, where the compose rules can use it in any feature-compatible position.

With these policies there is a 1-1 correspondence between parses and derivations. In fact, since all variables are instantiated after all subsitutions have applied, we can get the LC parser to construct an explicit representation of the corresponding derivation tree simply by adding tree arguments to the syntactic features of any grammar, as in (Pereira and Shieber, 1987, §6.1.2). For example, we can augment G1 with derivation tree arguments as follows, writing R/L for trees where R is root and L a list of subtrees, where • is merge and ∘ is move, and single capital letters are variables:

```
ϵ :: =v(V)  c(•/[ϵ::=v c/[],V])
ϵ :: =v(V)  +wh  c(∘/[•/[ϵ::=v c/[],V]])
knows :: =c(C)  =d(D)  v(•/[•/[knows::=c =d v/[],C],D])
likes :: =d(E)  =d(D)  v(•/[•/[likes::=d =d v/[],E],D])
Aca :: d(Aca::d/[])
Bibi :: d(Bibi::d/[])
what :: d(what::d -wh/[])  -wh
```

Without any change in the LC method above, with this grammar, the final start category in the last step of the LC parse of *Aca knows what Bibi likes* will have as its argument an explicit representation of the derivation tree of Figure 1, but with binary internal nodes replaced by • and unary ones by ∘.

A slightly different version of G1 will build the the derived X-bar tree for the example in Figure 1, or any other string in the infinite language of G1:

```
ϵ :: =v(V)  c(cP/[c/[ϵ/[],V]])
ϵ :: =v(V)  +wh(W)  c(cP/[W,c'/[c/[ϵ/[]],V]])
knows :: =c(C)  =d(D)  v(vP/[D,v'/[v/[knows/[]],C]])
likes :: =d(E)  =d(D)  v(vP/[D,v'/[v/[likes/[]],E]])
Aca :: d(dP/[Aca/[]])
Bibi :: d(dP/[Bibi/[]])
what :: d(dP(I)/[])  -wh(dP(I)/[what/[]])
```

Notice how this representation of the grammar uses a variable I to coindex the moved element with its original position. In the X-bar tree of Figure 1, that variable is instantiated to 0. Note also how the variable W gets bound to the moved element, so that it appears in under cP, that is, where the moving constituent 'lands'. See e.g. Stabler (2013, Appendix B) for an accessible discussion of how this kind of X-bar structure is related to the derivation, and see Kobele et al. (2007) for technical details. (See footnote 2 for an implementation of the approach presented here.)

## 6 Conclusions and future work

This paper defines left-corner MG parsing. It is non-deterministic, leaving the question of how to search for a parse. As in context free LC parsing, when there are empty left corners, backtracking search is not guaranteed to terminate. So we could use memoization or a beam or both. All of these search strategies are improved by discarding intermediate results which cannot contribute

to a completed parse, and so we define a very simple oracle which does this. That oracle suffices to make backtrack LC parsing of G1 and G2 feasible (see footnote 2). For grammars with empty left corners, stronger oracles can also be formulated, e.g. fully specifying all features and testing spans for emptiness. But for empty left corners, probably the left corner parser is not the best choice. Other ways of mixing top-down and bottom-up can be developed too, for the whole range of generalized left corner methods (Demers, 1977), some of which might be more appropriate for models of human parsing than LC (Johnson and Roark, 2000; Hale, 2014).

As noted earlier, Hunter (2017) aims to define a parser that appropriately models certain aspects of human sentence parsing. In particular, there is some evidence that, in hearing or reading a sentence from beginning to end, humans are inclined to assume that movements are as short as possible – "active gap-filling". It looks like the present model has a structure which would allow for modeling this preference in something like the way Hunter proposes, but we have not tried to capture that or any other human preferences here. Our goal here has been just to design a simple left-corner mechanism that does exactly what an arbitrary MG requires. Returning to Hunter's project with this simpler model will hopefully contribute to the project of moving toward more reasonable models of human linguistics performance.

There are many other natural extensions of these ideas:

- The proposed definition of LC parsing is designed to make correctness transparent, but now that the idea is clear, some simplifications will be possible. In particular, it should be possible to eliminate explicit unification, and to eliminate spans in stack elements.

- The LC parser could also be extended to other types of MG rules proposed for head-movement, adjunction, coordination, copying, etc. (Torr and Stabler, 2016; Fowlie, 2014; Gärtner and Michaelis, 2010; Kobele, 2006).

- Our LC method could also be adapted to multiple context free grammars (MCFGs) which are expressively equivalent, and to other closely related systems (Seki et al., 1991; Kallmeyer, 2010).

- Stanojević (2017) shows how bottom-up transition-based parsers can be provided for MGs, and those allow LSTMs and other neural systems to be trained as oracles (Lewis et al., 2016). It would be interesting to explore similar oracles for slightly more predictive methods like LC, and trained on recently built MGbank (Torr, 2018).

- For her 'geometric' neural realizations of MG derivations (Gerth and beim Graben, 2012), Gerth (2015, p.78) says she would have used an LC MG parser in her neural modeling if one had been available, so that kind of project could be revisited.

We leave these to future work.

## Acknowledgments

The first author is supported by ERC H2020 Advanced Fellowship GA 742137 SEMANTAX grant. The authors are grateful to Tim Hunter for sharing the early version of his LC paper and to Mark Steedman for support in developing and presenting this work.

## References

Steven P. Abney and Mark Johnson. 1991. Memory requirements and local ambiguities of parsing strategies. *Journal of Psycholinguistic Research* 20:233–249.

Alfred V. Aho and Jeffrey D. Ullman. 1972. *The Theory of Parsing, Translation, and Compiling. Volume 1: Parsing*. Prentice-Hall, Englewood Cliffs, New Jersey.

Joan Bresnan, Ronald M. Kaplan, Stanley Peters, and Annie Zaenen. 1982. Cross-serial dependencies in Dutch. *Linguistic Inquiry* 13(4):613–635.

Noam Chomsky. 1995. *The Minimalist Program*. MIT Press, Cambridge, Massachusetts.

Alan J. Demers. 1977. Generalized left corner parsing. In *4th Annual ACM Symposium on Principles of Programming Languages*. pages 170–181.

Hans den Besten and Gert Webelhuth. 1990. Stranding. In G. Grewendorf and W. Sternefeld, editors, *Scrambling and Barriers*, Academic Press, NY.

Victor J. Díaz, Vicente Carillo, and Miguel A. Alonso. 2002. A left corner parser for tree adjoining grammars. In *Proceedings of the Sixth International Workshop on Tree Adjoining Grammar and Related Frameworks (TAG+6)*. Association

for Computational Linguistics, pages 90–95. http://www.aclweb.org/anthology/W02-2213.

Meaghan Fowlie. 2014. Adjunction and minimalist grammars. In G. Morrill, R. Muskens, R. Osswald, and F. Richter, editors, *Formal Grammar: 19th International Conference*. Springer, NY, pages 34–51.

Hans-Martin Gärtner and Jens Michaelis. 2010. On the treatment of multiple wh-interrogatives in minimalist grammars. In Thomas Hanneforth and Gisbert Fanselow, editors, *Language and Logos: Studies in Theoretical and Computational Linguistics*, Academie Verlag, Berlin, pages 339–366.

Sabrina Gerth. 2015. *Memory limitations in sentence comprehension*. Universitätsverlag, Potsdam.

Sabrina Gerth and Peter beim Graben. 2012. Geometric representations for minimalist grammars. *Journal of Logic, Language and Information* 21(4):393–432.

Thomas Graf, Alëna Aksënova, and Aniello De Santo. 2016. A single movement normal form for minimalist grammars. In A. Foret, G. Morrill, R. Muskens, R. Osswald, and S. Pogodalla, editors, *Formal Grammar: 20th and 21st International Conferences, Revised Selected Papers*. Springer, Berlin, LNCS 9804, pages 200–215.

Günther Grewendorf, editor. 2015. *Remnant Movement*. Mouton de Gruyter, NY.

John T. Hale. 2014. *Automaton Theories of Human Sentence Comprehension*. CSLI, Stanford.

Henk Harkema. 2001a. A characterization of minimalist languages. In P. de Groote, G. Morrill, and C. Retoré, editors, *Logical Aspects of Computational Linguistics*. Springer, NY, LNCS 2099, pages 193–211.

Henk Harkema. 2001b. *Parsing Minimalist Languages*. Ph.D. thesis, University of California, Los Angeles.

Roland Hinterhölzl. 2006. *Scrambling, Remnant Movement, and Restructuring in West Germanic*. Oxford University Press, NY.

Tim Hunter. 2017. Left-corner parsing of minimalist grammars. Technical report, UCLA. Forthcoming.

Mark Johnson and Brian Roark. 2000. Compact non-left-recursive grammars using the selective left-corner transform and factoring. In *Proceedings of the 18th International Conference on Computational Linguistics, COLING*. pages 355–361.

Laura Kallmeyer. 2010. *Parsing Beyond Context-Free Grammars*. Springer, NY.

Makoto Kanazawa. 2008. A prefix correct Earley recognizer for multiple context free grammars. In *Proceedings of the 9th International Workshop on Tree Adjoining Grammars and Related Formalisms*. pages 49–56.

Richard S. Kayne. 1994. *The Antisymmetry of Syntax*. MIT Press, Cambridge, Massachusetts.

Gregory M. Kobele. 2006. *Generating Copies: An Investigation into Structural Identity in Language and Grammar*. Ph.D. thesis, UCLA.

Gregory M. Kobele. 2010. Without remnant movement, MGs are context-free. In C. Ebert, G. Jäger, and J. Michaelis, editors, *Mathematics of Language 10/11*. Springer, NY, LNCS 6149, pages 160–173.

Gregory M. Kobele, Christian Retoré, and Sylvain Salvati. 2007. An automata-theoretic approach to minimalism. In James Rogers and Stephan Kepser, editors, *Model Theoretic Syntax at 10. ESSLLI'07 Workshop Proceedings*.

Hilda Koopman and Anna Szabolcsi. 2000. *Verbal Complexes*. MIT Press, Cambridge, Massachusetts.

Mike Lewis, Kenton Lee, and Luke Zettlemoyer. 2016. LSTM CCG parsing. In *Proceedings of the 15th Annual Conference of the North American Chapter of the Association for Computational Linguistics*.

John W. Lloyd. 1987. *Foundations of Logic Programming*. Springer, Berlin.

Christopher D. Manning and Bob Carpenter. 1997. Probabilistic parsing using left corner language models. In *Proceedings of the 1997 International Workshop on Parsing Technologies*. Reprinted in H. Bunt and A. Nijholt (eds.) *Advances in Probabilistic and Other Parsing Technologies*, Boston, Kluwer: pp. 105-124. Also available as arXiv:cmp-lg/9711003.

Jens Michaelis. 2001. Transforming linear context free rewriting systems into minimalist grammars. In P. de Groote, G. Morrill, and C. Retoré, editors, *Logical Aspects of Computational Linguistics*. Springer, NY, LNCS 2099, pages 228–244.

Fernando C. N. Pereira and Stuart M. Shieber. 1987. *Prolog and Natural Language Analysis*. CSLI, Stanford.

Philip Resnik. 1992. Left-corner parsing and psychological plausibility. In *Proceedings of the 14th International Conference on Computational Linguistics, COLING 92*. pages 191–197.

D. J. Rosenkrantz and P. M. Lewis. 1970. Deterministic left corner parsing. In *IEEE Conference Record of the 11th Annual Symposium on Switching and Automata Theory*. pages 139–152.

Hiroyuki Seki, Takashi Matsumura, Mamoru Fujii, and Tadao Kasami. 1991. On multiple context-free grammars. *Theoretical Computer Science* 88:191–229.

Stuart M. Shieber. 1985. Evidence against the context-freeness of natural language. *Linguistics and Philosophy* 8(3):333–344.

Edward P. Stabler. 1997. Derivational minimalism. In C. Retoré, editor, *Logical Aspects of Computational Linguistics*, Springer-Verlag, NY, LNCS 1328, pages 68–95.

Edward P. Stabler. 2011. Computational perspectives on minimalism. In Cedric Boeckx, editor, *Oxford Handbook of Linguistic Minimalism*, Oxford University Press, Oxford, pages 617–641.

Edward P. Stabler. 2013. Two models of minimalist, incremental syntactic analysis. *Topics in Cognitive Science* 5(3):611–633.

Miloš Stanojević. 2017. Minimalist grammar transition-based parsing. In *International Conference on Logical Aspects of Computational Linguistics, LACL*. Springer, LNCS 10054, pages 273–290.

Mark J. Steedman. 2014. Categorial grammar. In A. Carnie, Y. Sato, and D. Siddiqi, editors, *Routledge Handbook of Syntax*, Routledge, NY, pages 670–701.

Terrance Swift and David S. Warren. 2012. XSB: Extending prolog with tabled logic programming. *Theory and Practice of Logic Programming* 12(1-2):157–187. ArXiv:1012.5123.

Gary Thoms and George Walkden. 2018. vP-fronting with and without remnant movement. *Journal of Linguistics* pages 1–54.

John Torr. 2018. Constraining mgbank: Agreement, l-selection and supertagging in minimalist grammars. In *Proceedings of the 56th Annual Meeting on Association for Computational Linguistics*. Association for Computational Linguistics, Melbourne, Australia.

John Torr and Edward Stabler. 2016. Coordination in minimalist grammars. In *Proceedings of the 12th Annual Workshop on Tree-Adjoining Grammars and Related Formalisms, TAG+*.

Gertjan van Noord. 1991. Head corner parsing for discontinuous constituency. In *Proceedings of the 29th Annual Meeting on Association for Computational Linguistics*. Association for Computational Linguistics, Stroudsburg, PA, USA, ACL 1991, pages 114–121.

Eric Wanner and Michael P. Maratsos. 1978. An ATN approach to comprehension. In M. Halle, J. Bresnan, and G. A. Miller, editors, *Linguistic Theory and Psychological Reality*, MIT Press, Cambridge, Massachusetts.