Minimalist Grammar Transition-Based Parsing

Miloš Stanojević(✉)

Institute for Logic, Language and Computation, University of Amsterdam, Amsterdam, The Netherlands
m.stanojevic@uva.nl

Abstract. Current chart-based parsers of Minimalist Grammars exhibit prohibitively high polynomial complexity that makes them unusable in practice. This paper presents a transition-based parser for Minimalist Grammars that approximately searches through the space of possible derivations by means of beam search, and does so very efficiently: the worst case complexity of building one derivation is $O(n^2)$ and the best case complexity is $O(n)$. This approximated inference can be guided by a trained probabilistic model that can condition on larger context than standard chart-based parsers. The transitions of the parser are very similar to the transitions of bottom-up shift-reduce parsers for Context-Free Grammars, with additional transitions for online reordering of words during parsing in order to make non-projective derivations projective.

Keywords: Minimalist Grammars · Shift-reduce parsing · Transition-based parsing · Swap transition · Two-stack automata

1 Introduction

Minimalist Grammar (MG) [14] is a formalization of Chomsky's Minimalist Program (MP) [4]. MG is one of the several grammar formalisms that go beyond Context-Free Grammars (CFG) in their expressive power (both in terms of weak and strong generative capacity). The main characteristic of MG is that constituents do not only combine to make bigger constituents, but they also can move during the course of derivation.

A standard derivation in Minimalist Program (and Minimalist Grammar) roughly looks like this: first we enumerate the words that are going to be used in the sentence with operation *select*; second, we combine operations *merge* and *move* in building the derivation bottom-up. The operation *merge* (sometimes called external merge) takes two constituents and puts them together. The *move* operation (sometimes called internal merge) takes a subtree and moves it upwards to the specifier position. So even though the words enter the derivation process in one order, by the end of the derivation they might form a completely different word order. This resembles the distinction between deep structure and surface structure from the early days of Generative Grammar [3]. The distinction between deep word order and surface word order does not exist in the Minimalist approach, but we will nevertheless adopt it here because it simplifies talking about some concepts.

M. Amblard et al. (Eds.): LACL 2016, LNCS 10054, pp. 273–290, 2016.
DOI: 10.1007/978-3-662-53826-5_17

Even though intuitively it might sound simple to build a recognizer (or parser) for a formalism that contains only two simple functions such as *merge* and *move* it turned out to be quite a difficult task. Early approaches [6,16] are based on bottom-up chart parsing which does an exhaustive search trough the space of all possible derivations. Because chart parsing is based on dynamic programming, such search is formally tractable in the sense of being polynomial. However, the polynomial complexity of chart parsing is still too high–$O(n^{4m+4})$ where n is the number of words in the sentence and m is the number of unique movement licensees present in the lexicon.

Transition-based parsers are an alternative to chart-based parsers. Transition-based parsers build the derivation step by step by using a set of well defined transitions that lead from one parsing state to the next one. Because they usually do not use dynamic programming, they cannot explore the full search space and that is why only approximate inference is possible. However, this did not stop transition-based parsers from matching and outperforming their chart-based counterparts in the area of CFG, CCG and dependency parsing both in terms of accuracy and in terms of speed [2,11,20]. Part of the reason for that is that dynamic programming in chart-based parsers requires from the probabilistic scoring model to condition only on the local context, while the transition-based parsers allow conditioning on any part of the derivation that was built. So giving up on exact inference allows us not only to gain in terms of speed but also it allows replacing weak probabilistic models with a much more powerful ones.

The top-down MG parser of Stabler [9,15] can be considered as an instantiation of transition-based parsing. It builds a minimalist derivation from top clause node c by *un-merging* and *un-moving* the nodes in the derivation recursively. Adding new operations to this parser requires finding a top-down equivalent of the minimalist operations that are traditionally defined in a bottom-up manner.

This paper presents a transition-based parser that is bottom-up, thus it does not require any changes in the definition or order of application of MG operations. It is similar to shift-reduce transition-based parsers, especially those that use a *swap* transition [8,11]. Just like majority of transition-based parsers, it employs no dynamic programming and employs approximate beam search of the space of derivations.

The main idea that motivates creation of this parser is based on observing that all MG derivation trees are projective trees with respect to the "deep word order". The displacement in the surface word order is a result of applying *move* operation. So if we would reorder the words in "the right way" then parsing should be projective and almost as easy as CFG parsing.

In the next three sections we present some background material for the transition-based bottom-up MG parser which covers definition of Minimalist Grammars, description of the existing chart parser for MGs and description of transition-based shift-reduce bottom-up parser for Context-Free Grammars. After that we fully specify the deductive system of the transition-based bottom-up minimalist parser and show some formal properties of it.

2 Minimalist Grammars

Here we describe a simple version of Minimalist Grammar as presented in [18] and [7]. This simple version does not deal with adjunction and head movement, but these extensions can easily be added to our parser and they do not influence the weak generative capacity of the MG [17].

A minimalist grammar G is a tuple $(\Sigma, Sel, Lic, Types, Lex, c, \mathscr{F})$, where

$\Sigma \neq \emptyset$ is an alphabet

Sel are "selecting features"

Lic are "licensing features"

Syn are "syntactic features" defined using Sel and Lic as a union of:
$selectors = \{=f | f \in Sel\}$
$selectees = \{\ \ f | f \in Sel\}$
$licensors = \{+f | f \in Lic\}$
$licensees = \{-f | f \in Lic\}$

$Types = \{::, :\}$ are the lexical type and the derived/phrasal type

$C = \Sigma^* \times Types \times Syn^*$ are "chains"

$Lex \subseteq C^+$ is a finite subset of chains with form $\Sigma^* \times \{::\} \times (selectors \cup licensors)^* \times selectees \times licensees^*$

$E = C^+$ are expressions

$c \in Sel$ is the feature used to define the *complete expression* $s : c$

$\mathscr{F} = \{merge, move\}$ are partial generating functions from E^* to E

merge: $(E \times E) \to E$ is a union of the following three functions, for $s, t \in \Sigma^*, \cdot \in \{:, ::\}$, $f \in Sel$, $\gamma \in Syn^*$, $\delta \in licensees^+$, and chains $\alpha_1, \ldots, \alpha_k, \iota_1, \ldots, \iota_l (0 \leq k,\ l)$:

$$merge1 \quad \frac{s :: =f\ \gamma \qquad\qquad t \cdot f, \alpha_1, \ldots, \alpha_k}{st : \gamma, \alpha_1, \ldots, \alpha_k}$$

$$merge2 \quad \frac{s : =f\ \gamma, \alpha_1, \ldots, \alpha_k \qquad\qquad t \cdot f, \iota_1, \ldots, \iota_l}{ts : \gamma, \alpha_1, \ldots, \alpha_k, \iota_1, \ldots, \iota_l}$$

$$merge3 \quad \frac{s \cdot =f\ \gamma, \alpha_1, \ldots, \alpha_k \qquad\qquad t \cdot f\ \delta, \iota_1, \ldots, \iota_l}{s : \gamma, \alpha_1, \ldots, \alpha_k, t : \delta, \iota_1, \ldots, \iota_l}$$

An illustration of these functions is presented in Fig. 1. In the figure expressions are represented as tree structures and on top of them is the list of unchecked features of the first chain. Chains that are waiting to move are represented as subtrees.

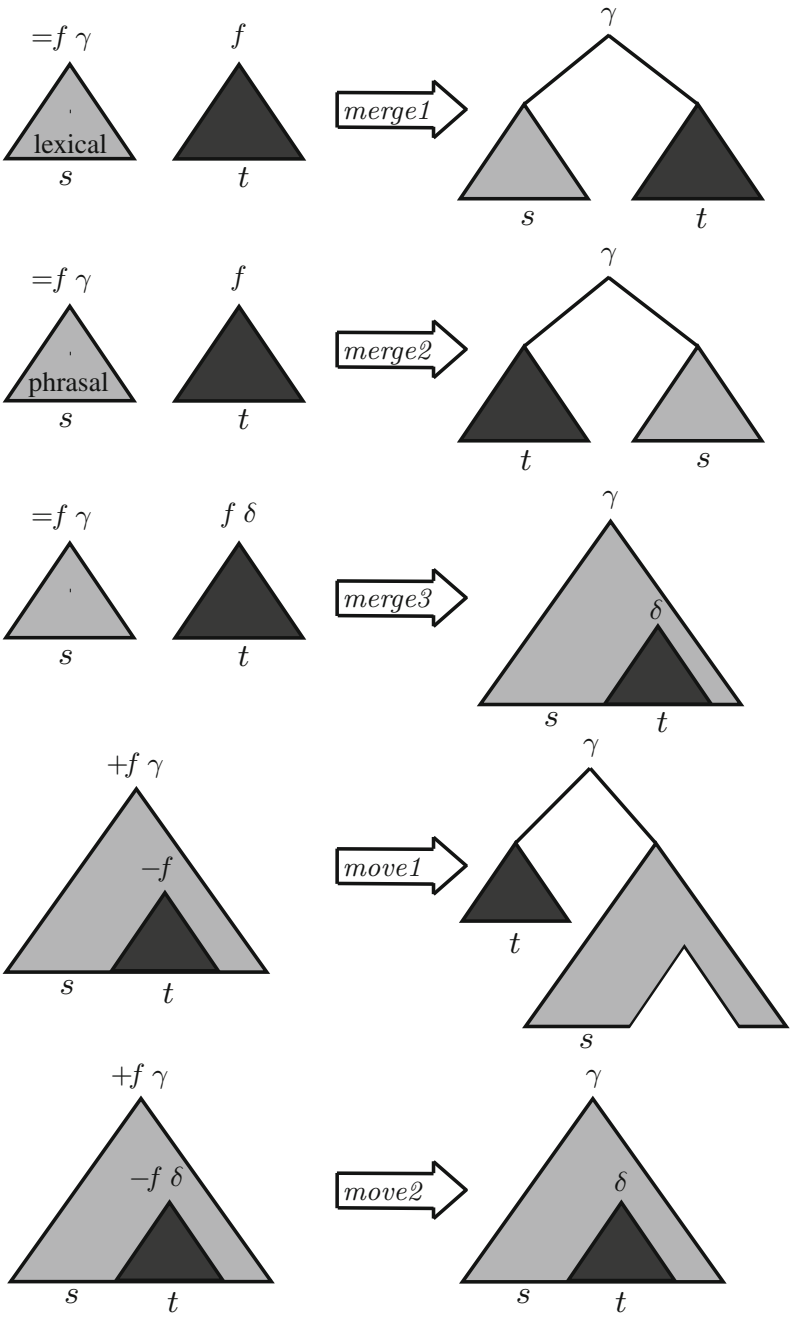

Fig. 1. Illustrations of Minimalist Grammar generating functions.

merge1 is combining the lexical head and its complement. The result is a new string in which string of the head s and the string of the complement t are concatenated and represented with st.

merge2 is combining a phrase that contains the head with the phrase that will be its specifier. Since specifier always comes on the left side, the resulting string is ts.

merge3 is combining phrases whose strings are not concatenated because the *licensees* δ will cause the phrase to move in the later steps of the derivation.

move: $E \rightarrow E$ is the union of the following two functions, for $s, t \in \Sigma^*$, $f \in Lic$, $\gamma \in Syn^*$, $\delta \in licensees^+$, and chains $\alpha_1, \ldots, \alpha_k, \iota_1, \ldots, \iota_l (0 \leq k, l)$ and SMC constraint defined bellow:

$$move1 \quad \frac{s : +f\, \gamma, \alpha_1, \ldots, \alpha_{i-1}, t : -f, \alpha_{i+1}, \ldots, \alpha_k}{ts : \gamma, \alpha_1, \ldots, \alpha_{i-1}, \alpha_{i+1}, \ldots, \alpha_k}$$

$$move2 \quad \frac{s : +f\, \gamma, \alpha_1, \ldots, \alpha_{i-1}, t : -f\, \delta, \alpha_{i+1}, \ldots, \alpha_k}{s : \gamma, \alpha_1, \ldots, \alpha_{i-1}, t : \delta, \alpha_{i+1}, \ldots, \alpha_k}$$

SMC is a simple version of "shortest move condition" [4]. It constrains the domain of *move* by not allowing any of $\alpha_1, \ldots, \alpha_{i-1}, \alpha_{i+1}, \ldots, \alpha_k$ to have licensee $-f$ as its first feature.

move1 is handling the movement of a subtree with yield t into a specifier of the current tree. Because *move1* moves subtree that is landing (it is not going to move any more) we can safely concatenate strings into ts.

move2 is handling the movement of a subtree that will continue moving in the later steps of the derivation because it has unchecked licensees δ, thus there is no need for concatenating strings s and t.

$CL(G)$ is a set of expressions generated by taking the closure over Lex and generating functions in $\mathscr{F}$.

$yield(e)$ is defined only over complete expressions $(s \cdot c)$ and is an alphabet component of the only chain in the expression e.

The language defined by the grammar G is $L(G) = \{s \mid \exists e \in CL(G) \wedge e \text{ is } s \cdot c\}$. In other words, a language defined by G is the set of yields of all complete expressions that are part of the closure of G.

3 Chart-Based Parser for MG

The first recognizers for MG were chart-based recognizers of Harkema [6] and Stabler [16]. The parsing strategy is presented in the form of deductive rules. These rules could be used as part of some closure computation engine, such as the ones based on "parsing as deduction" [13], in order to get efficient inference by using dynamic programming.

The general idea of "parsing as deduction" [13] is that we are trying to prove that the sentence that is parsed is part of the language defined by the grammar. We start with some claims that do not require proving i.e. axioms (for example "from position i till position i+1 there is a word w_i), and after that we

apply deductive rules recursively until we prove the goal statement (for example "sentence with words $w_0, \ldots, w_{n-1}$ has only c as an unchecked feature") or until we exhaust all possibilities without managing to prove that the sentence is part of the language.

The statements that the deduction engine is working with are encoded in the form of "items". At any step of the parsing process, all the items can be divided in two groups: items that can trigger further deduction and items that are proved but they do not trigger future deduction. Items of the first group are stored in a queue called *agenda* and the items of the second group are stored in a data structure for efficient retrieval that is called *chart*. With this terminology we can say that the parsing process starts with putting axiomatic items in agenda and applying deduction rules on all of them in order. If the result of a deduction rule can trigger future deduction, it is added both to the chart and to the agenda, otherwise it is added only to the chart.

Items of the minimalist chart parser are essentially encodings of MG expressions which instead of using strings of alphabet use ranges of covered words in the sentence. So for example, item $(2, 5) := n\ v$ can be read as "this is a phrase with features $=n$ and v and it covers continuous span of words from positions 2 until position 5 in the observed word order".

With that interpretation the following deduction rules have been proven to be sound and complete [6], where n is the length of the sentence, w_i is word at position i and i can go between 0 and n:

$$\textit{axiom}\quad (i, i+1) :: \alpha \quad s.t. \quad w_i :: \alpha \in \mathit{Lex}$$

$$\textit{axiomEpsilon}\quad (i, i) :: \alpha \quad s.t. \quad \varepsilon :: \alpha \in \mathit{Lex}$$

$$\textit{goal}\quad (0, n) \cdot c$$

$$\textit{merge1}\ \frac{(a, b) :: =f\ \gamma \qquad (b, c) \cdot f, \alpha_1, \ldots, \alpha_k}{(a, c) : \gamma, \alpha_1, \ldots, \alpha_k}$$

$$\textit{merge2}\ \frac{(b, c) : =f\ \gamma, \alpha_1, \ldots, \alpha_k \qquad (a, b) \cdot f, \iota_1, \ldots, \iota_l}{(a, c) : \gamma, \alpha_1, \ldots, \alpha_k, \iota_1, \ldots, \iota_l}$$

$$\textit{merge3}\ \frac{(a, b) \cdot =f\ \gamma, \alpha_1, \ldots, \alpha_k \qquad (c, d) \cdot f\delta, \iota_1, \ldots, \iota_l}{(a, b) : \gamma, \alpha_1, \ldots, \alpha_k, (c, d) : \delta, \iota_1, \ldots, \iota_l}$$

$$\textit{move1}\ \frac{(b, c) : +f\ \gamma, \alpha_1, \ldots, \alpha_{i-1}, (a, b) : -f, \alpha_{i+1}, \ldots, \alpha_k}{(a, c) : \gamma, \alpha_1, \ldots, \alpha_{i-1}, \alpha_{i+1}, \ldots, \alpha_k}$$

$$\textit{move2}\ \frac{(a, b) : +f\ \gamma, \alpha_1, \ldots, \alpha_{i-1}, (c, d) : -f\ \delta, \alpha_{i+1}, \ldots, \alpha_k}{(a, b) : \gamma, \alpha_1, \ldots, \alpha_{i-1}, (c, d) : \delta, \alpha_{i+1}, \ldots, \alpha_k}$$

Naturally, *move* is subject to SMC constraint.

4 Transition-Based Bottom-Up Parser for CFG

Before we move to the formal description of the transition-based Minimalist parser, we will make a small digression and informally present a type of

shift-reduce parser for CFG in Chomsky Normal Form (CNF) similar to the one presented in [10,12]. This algorithm is used as a basis on which the Minimalist transition-based parser is built.

The state of the transition based parser is usually called *configuration*. A configuration consists of two data structures: stack σ and buffer (usually implemented as a queue) β. For CFG in CNF the initial configuration is an empty stack and the buffer filled with words of the sentence that is parsed. Shift action removes the first word in the current buffer and puts its POS tag on top of the stack. Reduce operation takes the two elements from top of the stack and produces a new element that goes back to the stack if there is a grammar rule that allows that.

The deduction rules are shown bellow, where [ ] represents an empty stack or an empty buffer, σ represents a stack, $\sigma|x$ represents a stack that is the result of pushing element x on top of stack σ, β represents buffer, $x|\beta$ represents a buffer (queue) with head x and tail β, G represents a CFG in CNF, w is a variable representing any word, X, Y, Z are variables representing any non-terminal, and S is the root non-terminal of the grammar G.

$$
\begin{aligned}
\textit{axiom} \quad & \langle [], [w_1, \ldots, w_n] \rangle \\
\textit{goal} \quad & \langle [S], [] \rangle \\
\textit{shift}\{X\} \quad & \frac{\langle \sigma, w|\beta \rangle}{\langle \sigma|X, \beta \rangle} \quad X \rightarrow w \in G \\
\textit{reduce} \quad & \frac{\langle \sigma|X|Y, \beta \rangle}{\langle \sigma|Z, \beta \rangle} \quad Z \rightarrow XY \in G
\end{aligned}
$$

Deriving the goal configuration can be done in several ways. One of them is by using a chart-based algorithm that would compute a full closure of these deduction rules over the axiomatic configuration. Another is a transition-based approach where the algorithm would treat each deduction rule as a transition and only a predefined number of high probability sequence of transitions will be explored. The computational complexity of the transition-based algorithm is $O(n)$ because the number of shift transitions is not bigger than the number of words in the sentence, and the same holds for the number of reduce transitions. This nice property of transition-based shift-reduce parsing has caused its wide adoption in the natural language processing community which produced many extensions and implementations of the transition systems for semantic parsing [19], CCG parsing [2], non-projective constituency parsing [8] and non-projective dependency parsing [11].

5 Transition-Based Bottom-Up Parser for MG

It is striking how many of the operations and structures from transition-based shift-reduce parsing have their counterparts in Minimalist Syntax, as described in [4] and formalized in [5]. The stack plays a similar role to a minimalist *workspace*,

a buffer looks similar to a *lexical array*, a configuration is like a *stage* in the minimalist derivation, shift behaves as a *select* operation and *reduce* behaves like *merge*.

A big part of Shift-Reduce parser can easily be modified to give support for Minimalist Grammars. Out of 5 rules of Minimalist Grammars, 4 are trivial to integrate: *move1* and *move2* are essentially unary feature simplification transitions, and *merge1* and *merge2* are operations that put two consecutive constituents together in almost the same way as CFG does. The only complicated cases are *merge3* and empty string terminals.

5.1 Handling Discontinuities with Online Reordering

Operation *merge3* causes complications because it merges discontinuous elements. To account for that, we introduce the possibility to reorder the elements on the stack so that the constituents that are not neighbouring can be merged. However, that is not enough because the non-head argument of *merge3* will later trigger one of the move operations that needs to satisfy neighbouring conditions. To be able to easily check if the moving constituent satisfies this constraint, the representations of the constituent that is used is the same as the representation of the constituent in the chart parser: spans and their associated chains.

5.2 Explicit Generation of Empty Strings

The problem of empty strings is mostly specific to Minimalist Grammars, since many grammar formalisms that do not have empty categories, for example CCG, do not need to account for it. Empty string terminals are introduced just like the non-empty string terminals by using an operation similar to shift, except that the buffer is not influenced by the transition. The representation of the shifted empty terminal is similar to the one in chart based parser, except that for the span we use wildcard symbols $(*, *)$ – what that means is that the constituent with this span is not a subject to the linear ordering constraints imposed by *merge1*, *merge2* and *move1*, but only to the feature matching constraints. The interpretation of the wildcard $(*, *)$ can depend on the operation that is being used in. For example, if we have a head with span $(2, 5)$ and it selects the empty constituent with span $(*, *)$ by using *merge1* then we can treat the empty constituent as if it is positioned at $(5, 5)$ (in the gap between 4th and 5th word). Note that the size of the span $(*, *)$ can never be bigger than 0 because it represents only empty elements.

Being able to explicitly generate empty strings can also cause the parser to generate empty strings *ad infinitum*, casing the parser to get stuck in the infinite loop. To prevent that we can define upper number of empty strings that can be generated for the sentence of length n which we allow to be any linear function of n. Knowing ahead of time which linear function correctly predicts the number of empty elements in a sentence is impossible because there might be no function that does that (there can be infinite number of empty strings). However, for the

actual natural languages the number of empty strings is not infinite. A heuristic that can be used to determine the maximal number of empty strings is the one which assumes that: (1) empty strings appear only with function words, (2) there is a some constant of maximal number of function projections per clause (for example based on hierarchy of projections [1]) and (3) every clause contains at least one pronounced word. In that case the maximal number of empty strings is the product of the maximal number of clauses (which is the number of observed) and the maximal number of function projections per clause. Clearly this method is too conservative about the upper bound of the number of empty strings, so in practice maybe a better approach would be to estimate the number on a treebank on which the parser is trained.

5.3 Parser Description

The basic units of the minimalist transition based parser are lexical items (LI) and minimalist items (MI). Lexical items are just indices of the words in the sentence that is being parsed. Minimalist items are the same as the items in the chart-based parser that was presented in Sect. 3. For clarity we surround minimalist items with braces.

The main control structures are two stacks σ_1 and σ_2 and one buffer β (implemented as a queue). Buffer β represents the sequence of lexical items waiting to be selected for building the derivation. Stacks are sequences of already built syntactic objects i.e. minimalist items. The first stack σ_1 is the main stack that is used for actual building of syntactic objects by application of *merge*, *move* and

$$
\begin{aligned}
\textit{axiom}\quad & \langle\ [\,],\quad [\,],\quad [0,\ldots,n-1],\quad 0\ \rangle \\
\textit{goal}\quad & \langle\ [\{(0,n)\cdot c\}],\quad [\,],\quad [\,],\quad k\ \rangle \\
\textit{select}\{\gamma\}\quad & \frac{\langle\ \sigma_1,\quad \sigma_2,\quad i|\beta,\quad k\ \rangle}{\langle\ \sigma_1|\{(i,i+1)::\gamma\},\quad \sigma_2,\quad \beta,\quad k\ \rangle}\ w_i{::}\gamma \in Lex \\
\textit{selectEpsilon}\{\gamma\}\quad & \frac{\langle\ \sigma_1,\quad \sigma_2,\quad \beta,\quad k\ \rangle}{\langle\ \sigma_1|\{(*,*)::\gamma\},\quad \sigma_2,\quad \beta,\quad k{+}1\ \rangle}\ k<e \wedge \varepsilon{::}\gamma \in Lex \\
\textit{tmerge}\quad & \frac{\langle\ \sigma_1|x|y,\quad \sigma_2,\quad \beta,\quad k\ \rangle}{\langle\ \sigma_1|merge(x,y),\quad \sigma_2,\quad \beta,\quad k\ \rangle}\ (x,y) \in Dom(merge) \\
\textit{tmove}\quad & \frac{\langle\ \sigma_1|x,\quad \sigma_2,\quad \beta,\quad k\ \rangle}{\langle\ \sigma_1|move(x),\quad \sigma_2,\quad \beta,\quad k\ \rangle}\ x \in Dom(move) \\
\textit{swap}\quad & \frac{\langle\ \sigma_1|x|y,\quad \sigma_2,\quad \beta,\quad k\ \rangle}{\langle\ \sigma_1|y,\quad x|\sigma_2,\quad \beta,\quad k\ \rangle}\ spanStart(x) < spanStart(y) \\
\textit{takeBack}\quad & \frac{\langle\ \sigma_1,\quad x|\sigma_2,\quad \beta,\quad k\ \rangle}{\langle\ \sigma_1|x,\quad \sigma_2,\quad \beta,\quad k\ \rangle}
\end{aligned}
$$

Fig. 2. Deduction system for MG transition-based parser

variants of *select* operations. The second stack σ_2 is the auxiliary stack that is used for reordering minimalist items in σ_1 (it will be explained later how). The configuration (parser state) consists of σ_1, σ_2, β and an integer k that represents the count of ε transitions (transitions that generate empty strings) that led to that configuration.

The deduction system of the minimalist transition based parser is shown in Fig. 2. The starting configuration of the parser is a configuration with empty stacks (no syntactic object is built so far), buffer filled with indices of words in the sentence and the count of ε transitions set to 0. The goal configuration that the parser tries to get to is the one in which all elements of the buffer would be used, there would be no elements on hold in the auxiliary stack and the main stack has only one MI which is the complete MI (as defined for the chart parser).

The *select*$\{\gamma\}$ transition takes the first LI in the buffer and puts it on top of the main stack in the form of MI with γ chain. That happens iff there is an entry in the lexicon where word represented by LI has chain γ. That can be done only for non-ε entries in the lexicon because these are the only entries that can be directly observed in the buffer. The ε entries in the lexicon are handled by *selectEpsilon*$\{\gamma\}$ transition which does not influence the buffer, but does increase the count of the empty strings and must respect the constraint that count should not be above some prespecified number e.

Naturally, we need a transition *tmerge* that uses minimalist operation merge and transition *tmove* that uses the minimalist operation move. These transitions are applied only if the logical expressions represented by the MI on top of the main stack fall in the domain of the functions *merge* and *move* (as defined in the chart-based parser).

Discontinuity can be achieved by reordering the words in the sentence in such a way that the sentence becomes contiguous. We illustrate this with the derivations of the sentence "Phong likes what Roki draws" with the following Minimalist Grammar:

$$\begin{aligned}
\varepsilon &:: {=}vc \\
\varepsilon &:: {=}v\ {+}whc \\
\text{likes} &:: {=}c\ {=}d\,v \\
\text{draws} &:: {=}d\ {=}d\,v \\
\text{Phong} &:: d \\
\text{Roki} &:: d \\
\text{what} &:: d\ {-}wh
\end{aligned}$$

The derived tree for this sentence is shown in Fig. 3a. The leaf nodes in this tree are ordered in the same way as is the surface word order of the sentence. The head for each constituent is marked with an arrow-like label, which points to the constituent which contains the head. In this derived tree it is not possible to see in which order and where the operations *merge* and *move* were applied. In order to see this we need a derivation tree like the one presented in Fig. 3b. In

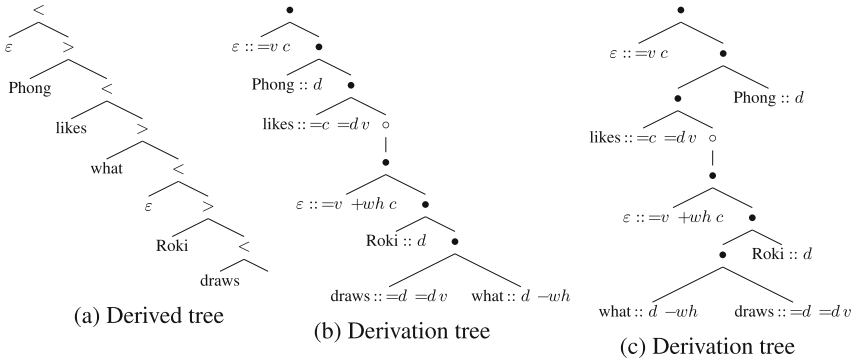

Fig. 3. Trees for sentence "Phong likes what Roki draws"

this tree, the merge operation is marked with • and the move operation with ∘. The derivation tree is a rooted unordered binary branching tree. The ordering of nodes does not matter because merge is a commutative operation. Hence, both Fig. 3b and c represent the same derivation tree which produces the derived tree in Fig. 3a.

If the words in the sentence that is being parsed were ordered the way leaves are ordered in Fig. 3b or c then parsing would be projective and the deduction rules we defined so far would suffice. There can be exponentially many permutations of the words that would make the parsing projective and it is enough if the parser finds only one of them. We call these orderings "deep word orders".

To achieve this reordering of the elements that are participating in parsing we introduce two transitions to the parser: *swap* and *takeBack*. The transition *swap* takes the 2nd top MI from the σ_1 and puts it on top of the auxiliary σ_2. The transition *takeBack* returns these MI back to the main stack. By combining *swap* and *takeBack* we can derive any permutation of the minimalist items of the main stack. To prevent cycles of *swap* and *takeBack* there is a constraint that starting positions of the MIs that are being swapped are in the original word order.

A full transition sequence for the example sentence and example grammar is given in Fig. 4. This is only one of the possible transition sequences. We could have chosen some other sequence of *swap* and *takeBack* transitions that would produce the same derivation tree. The key part of this example are transitions *swap* and *takeBack*. These two transitions swap the order of minimalist items for the words "what" and "Roki" and in this way make parsing projective.

stage	Main stack σ_1	Auxiliary stack σ_2	Buffer β	k	transition
1			0 , 1 , 2 , 3 , 4 | | | | | Phong likes what Roki draws	0	*select*{**d**}
2	$\{(0,1) :: d\}$ | Phong		1 , 2 , 3 , 4 | | | | likes what Roki draws	0	*select*{**=c =d v**}
3	$\{(0,1) :: d\}, \{(1,2) :: =c\ =d\ v\}$ | | Phong likes		2 , 3 , 4 | | | what Roki draws	0	*select*{**d −wh**}
4	$\{(0,1) :: d\}, \{(1,2) :: =c\ =d\ v\}, \{(2,3) :: d\ -wh\}$ | | | Phong likes what		3 , 4 | | Roki draws	0	*select*{**d**}
5	$\{(0,1) :: d\}, \{(1,2) :: =c\ =d\ v\}, \{(2,3) :: d\ -wh\}, \{(3,4) :: d\}$ | | | | Phong likes what Roki		4 | draws	0	*swap*
6	$\{(0,1) :: d\}, \{(1,2) :: =c\ =d\ v\}, \{(3,4) :: d\}$ | | | Phong likes Roki	$\{(2,3) :: d\ -wh\}$ | what	4 | draws	0	*takeBack*
7	$\{(0,1) :: d\}, \{(1,2) :: =c\ =d\ v\}, \{(3,4) :: d\}, \{(2,3) :: d\ -wh\}$ | | | | Phong likes Roki what		4 | draws	0	*select*{**=d =d v**}
8	$\{(0,1) :: d\}, \{(1,2) :: =c\ =d\ v\}, \{(3,4) :: d\}, \{(2,3) :: \mathbf{d}\ -wh\}, \{(4,5) :: \mathbf{=d}\ =d\ v\}$ | | | | | Phong likes Roki what draws			0	*tmerge*
9	$\{(0,1) :: d\}, \{(1,2) :: =c\ =d\ v\}, \{(3,4) :: \mathbf{d}\}, \{(4,5) :: \mathbf{=d}\ v,\ (2,3) :: -wh\}$ | | | | Phong likes Roki • what draws			0	*tmerge*
10	$\{(0,1) :: d\}, \{(1,2) :: =c\ =d\ v\}, \{(3,5) :: v,\ (2,3) :: -wh\}$ | | | Phong likes • Roki • what draws			0	*selectEpsilon*{**=v +wh c**}

Fig. 4. Part 1 of example transition sequence

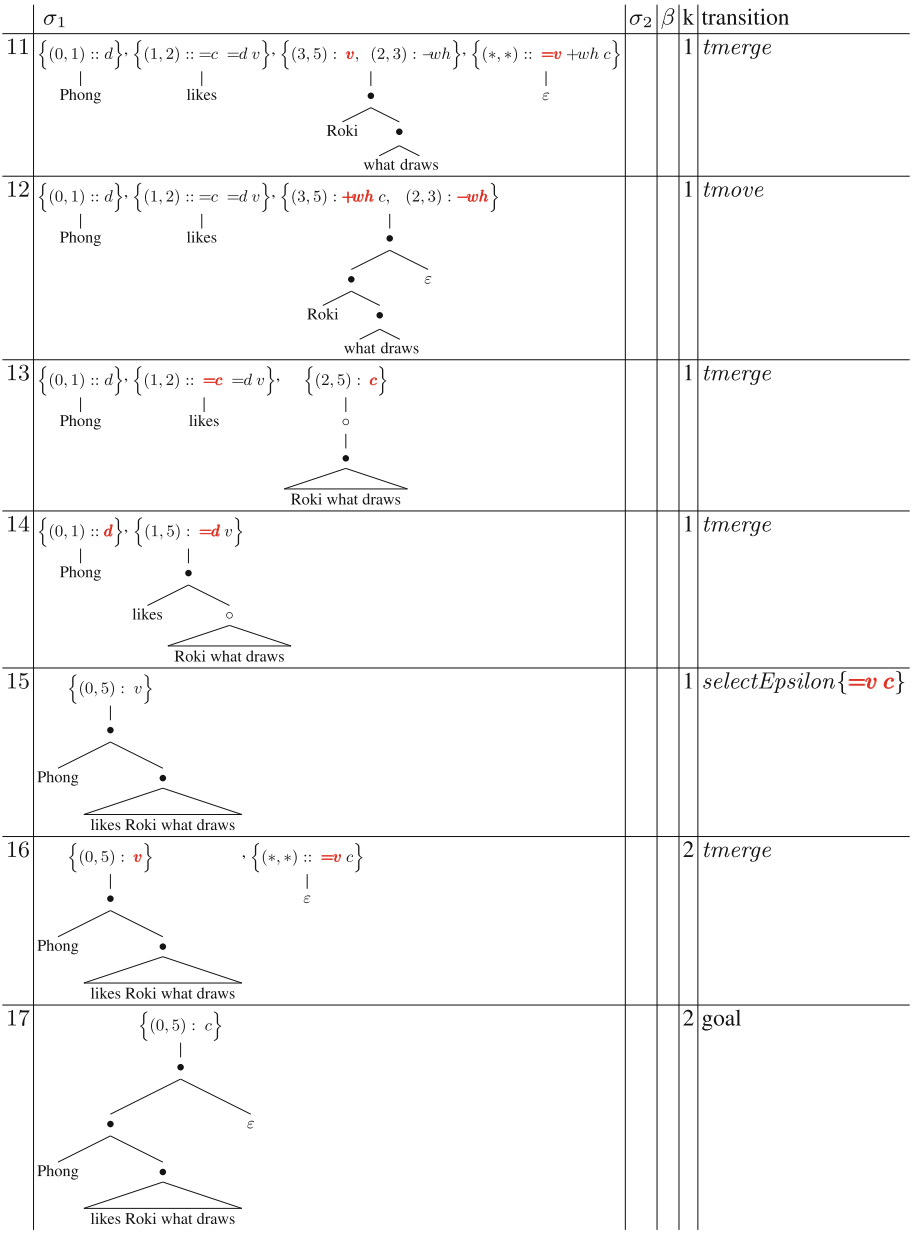

Fig. 5. Part 2 of example transition sequence

6 Soundness, Completeness and Complexity

Here we give sketches of the proofs for soundness, completeness and complexity of the transition-based algorithm. The proofs rely in big part on proofs of soundness and complexity of Harkema's Minimalist chart parser [6] (presented in Sect. 3) because the transition-based parser and Harkema's parser have isomorphic structure of items and operations over them.

6.1 Soundness

Proving soundness is trivial. The only part of our system that gives logical claims about the sentence are minimalist items and they have the same form and semantics as items in Harkema's chart parser. All the transitions that modify these MIs have their equivalent in Harkema's parser. The transitions *select*$\{\gamma\}$ and *selectEpsilon*$\{\gamma\}$ bijectivelly map to axiomatic rules of Harkema's parser while transitions *tmerge* and *tmove* directly call the corresponding Harkema's definitions of *merge* and *move* that were presented in Sect. 3. The transitions *swap* and *takeBack* do not modify the *mini-items* so they do not influence the soundness of the algorithm.

The deduction system presented in this paper is isomorphic to that of Harkema's parser. Consequently, every item reachable by the transition-based parser is also reachable by Harkema's parser. Since all items generated by Harkema's parser are sound, it follows that all of the items generated by the transition-based parser are sound too.

6.2 Completeness and Construction of an Oracle

Even though the deduction systems are isomorphic in terms of items and operations over them, that does not entail that the set of items that can be generated is equivalent. Harkema's parser is proven to be complete – it can generate all the possible sound items by starting with the axiom and then applying the deduction rules until no new items can be generated. However, the transition-based parser has three major constraints.

The first one is that it is approximate – it will explore only the part of the search space that is considered the most probable by the scoring model. Obviously, this depends on the quality of the scoring model, thus for the sake of the proof, we will assume the parser has a beam of unbounded size. In other words, let us assume an exhaustive search where no item is pruned however poorly scored. Our goal is to prove that with the perfect scoring model the right derivation will be found by the transition system.

The second main difference is that operations can be applied only to the top elements of the main stack, unlike Harkema's parser which can apply operations to any two items that have been derived (it has global access to its "workspace", which is a chart). The main question is then whether this limits the set of items that can be deduced using the transition-based system. Given that Harkema's parser is complete and that all functions of Harkema's parser are present in

the transition-based parser, we just need to show that for any derivation tree there is a sequence of transitions that would derive it. This conversion of a MG derivation to the sequence of transitions can be interpreted as a construction of the *oracle* sequence of transitions. The oracle is used often in transition-based parsing as a sequence of transitions on which the probabilistic parsing model is trained. There are many possible oracles for any MG tree so in the probabilistic setting all these oracles should ideally be treated as latent variables. However, experience from other grammar formalisms shows that using just one oracle seems to be good enough for most of the parsers.

The third difference is that all empty strings are explicitly generated in the transition-based parser while in the chart-based parser infinite number of empty strings can be compactly represented thanks to the dynamic programming. Since the maximal number of empty strings that can be generated is limited by some predefined constant e, any proof of completeness is limited to the trees that have less than e empty strings. Here we will assume that e is infinite. In other words, we show that for a sufficiently large e any MG derivation can be generated.

First, we cover the case in which the words of the sentence are in one of the many possible "deep word orders" (word orders in which the derivation tree is projective). In this case extracting the sequence of transitions is easy: we just need to traverse the derivation tree in the post-order traversal (the "order" of the subtrees is based on the deep order of words). Every time we encounter a leaf in the derivation tree, it will cause a $select\{\gamma\}$ or $selectEpsilon\{\gamma\}$ transition. Every time we encounter a binary branching node, it will be a *tmerge* transition and every time we encounter a unary branching node, it will be a *tmove* transition. So, if the words are processed in the projective "deep order" there is always a transition sequence that will produce any projective derivation tree.

Now we show that even if the words are in non-projective word order that they could still be processed in a projective order. Let us say that the next LI that should come on the main stack is on the m^{th} position in the buffer. What we need to do is m $select\{\gamma\}$ transitions, followed by $m-1$ *swap* transitions. The alternative situation is that the next element is on the m^{th} positions in the auxiliary stack σ_2. The process is the same except that instead of invoking m $select\{\gamma\}$ transitions we invoke m *takeBack* transitions, followed by $m-1$ *swap* transitions.

Given that we can find transitions for any derivation in projective word order, and that any non-projective derivation can be traversed in projective order, it follows that we can find a transition sequence for any non-projective derivation. This, together with Harkema's proof of the completeness of the basic functions *merge* and *move* that the transition-based parser uses, makes the transition-based parser complete.

6.3 Computational Complexity

Because the transition-based parser does not pack its derivations by using dynamic programming, its complexity with unbounded beam will be exponential in sentence length. However, since transition-based parsers are never used

to search trough the full space of derivations, but always with a limited beam, we will here focus on the complexity of constructing a single derivation.

The complexity can be determined by estimating the maximal number of times each transition type will be used. The number of transitions *select*$\{\gamma\}$ is n because it will be used only once and for each of the observed words. By design, the transition *selectEpsilon*$\{\gamma\}$ will be used maximally e times which is a linear function of n. The maximal number of *tmerge* transitions is equivalent to the maximal number of binary nodes in a binary branching tree over a $n+e$ words which is $n+e-1$. The maximal number of *tmove* operations is equivalent to the number of all the *licensees* in the sentence. If the maximal number of licensees that the lexicon has per entry is some constant m, then the maximal number of licensees in the sentence is $m*(n+e)$. The number of *swap* operations is equivalent to the number of *takeBack* operations. In the best case there will be no swapping (all the words are in one of the possible deep word orders), and in the worst case there will be in, asymptotic notations, $O(n^2)$ *swap*s and *takeBack*s. Since m is constants we can say that for all operations the asymptotic complexity is $O(n)$, except for swapping transitions that can be between $O(n)$ and $O(n^2)$. So, the total complexity of building one derivation is dependent on the *swap* and *takeBack* transitions making this parser's worst case complexity $O(n^2)$ and best case complexity $O(n)$.[1]

7 Parsing of Finite-State Automaton

The minimalist transition-based parser can easily be extended to parse not only sentences, but also regular sets of sentences encoded in a finite-state automaton (FSA). All that needs to be modified are representations of buffer and the *select*$\{\gamma\}$ transition since it is the only transition that changes the buffer. The buffer would now instead of a queue be an FSA that is being parsed and it would additionally contain the pointer to *the current state* up until which the input was consumed. The new *select*$\{\gamma\}$ transition would pick one of the outgoing arcs of the current state, consume the arc's label (the same way it used to consume the word in the buffer) and change the current state to the target state of the selected arc. The initial (axiom) configuration would be with empty stacks and with buffer FSA that has its current state set to its initial state. The goal configuration would be the same as before, except for having the additional condition that the current state in an FSA is the final state of an FSA.

The simplicity of doing discontinuous parsing of FSAs can be crucial is some cases when the input is ambiguous. Take for instance morphologically rich languages: doing morphological segmentation in these languages is difficult and the selection of the right segmentation can be done only during the syntactic processing because of different forms of agreement. Now instead of parsing a potentially bad 1-best guess of the morphological analyser, we can take a full lattice that

[1] Interestingly, Nivre shows that the number of swap transitions in the real dataset for dependency parsing is very small which makes the transition-based parser run in expected linear time [11]. Hopefully, this will also be true for Minimalist Grammars.

would encode many hypotheses of the possible segmentation and then let the parser decide which one is the best. Another use case of FSA parsing is processing the ambiguous output of the speech recognizer which is often encoded in lattices.

8 Conclusion and Future Work

The transition-based parser presented in this paper, if supported by a good probabilistic scoring model, could handle even the longest sentences very efficiently. The very small computational complexity of building one derivation makes the transition-based parser for Minimalist Grammars as fast as its counterparts for CCG, dependency and constituency parsing.

The usual motivation for using simpler formalisms such as dependency and context-free grammars is their efficiency. However, given that the presented parser is asymptotically as fast as the approximate parsers for the simpler formalisms, the natural language processing community can start considering Minimalist Grammars as a possible more expressive alternative. In order for this transition to become a reality, a necessary next step is creation of the scoring model, as well as the creation of a Minimalist treebank on which the scoring model would be trained.

Acknowledgments. I would like to thank Raquel G. Alhama, Joachim Daiber, Phong Le, Wilker Aziz and Khalil Sima'an for useful discussions and comments on the early versions of this paper. I am also grateful to the three anonymous reviewers for their insightful comments. This work was supported by STW grant nr. 12271.

References

1. Adger, D.: Core Syntax: a Minimalist Approach, vol. 33. Oxford University Press, Oxford (2003)
2. Ambati, B.R., Deoskar, T., Johnson, M., Steedman, M.: An incremental algorithm for transition-based CCG parsing. In: Proceedings of 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies. Association for Computational Linguistics (2015)
3. Chomsky, N.: Syntactic Structures. Mouton & Co., The Hague (1957)
4. Chomsky, N.: The Minimalist Program, vol. 1765. Cambridge University Press, Cambridge (1995)
5. Collins, C., Stabler, E.: A formalization of minimalist syntax. Syntax **19**(1), 43–78 (2016)
6. Harkema, H.: A recognizer for minimalist languages. In: Bunt, H., Carroll, J., Satta, G. (eds.) New Developments in Parsing Technology. TSLT, vol. 23, pp. 251–268. Springer, Dordrecht (2005)
7. Hunter, T., Dyer, C.: Distributions on Minimalist Grammar derivations. In: Proceedings of 13th Meeting on the Mathematics of Language (MoL 2013), pp. 1–11. Association for Computational Linguistics, Sofia, August 2013

8. Maier, W.: Discontinuous incremental shift-reduce parsing. In: Proceedings of 53rd ACL (Long Papers), vol. 1, pp. 1202–1212. Association for Computational Linguistics, Beijing, July 2015
9. Mainguy, T.: A probabilistic top-down parser for Minimalist Grammars (2010). CoRR arXiv:abs/1010.1826
10. Mi, H., Huang, L.: Shift-reduce constituency parsing with dynamic programming and POS tag Lattice. In: Proceedings of 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, pp. 1030–1035. Association for Computational Linguistics, Denver, May–June 2015
11. Nivre, J.: Non-projective dependency parsing in expected linear time. In: Proceedings of Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP, pp. 351–359 (2009)
12. Sagae, K., Lavie, A.: A classifier-based parser with linear run-time complexity. In: Proceedings of 9th International Workshop on Parsing Technology, Parsing 2005, pp. 125–132. Association for Computational Linguistics., Stroudsburg (2005)
13. Shieber, S.M., Schabes, Y., Pereira, F.C.N.: Principles and implementation of deductive parsing. J. Log. Program. **24**, 3–36 (1995)
14. Stabler, E.: Derivational minimalism. In: Retoré, C. (ed.) LACL 1996. LNCS, vol. 1328, pp. 68–95. Springer, Heidelberg (1997). doi:10.1007/BFb0052152
15. Stabler, E.: Top-down recognizers for MCFGs and MGs. In: Proceedings of 2nd Workshop on Cognitive Modeling and Computational Linguistics, pp. 39–48. Association for Computational Linguistics, Portland, June 2011
16. Stabler, E.P.: Minimalist Grammars and recognition. In: Rohrer, C., Rossdeutscher, A., Kamp, H. (eds.) Linguistic Form and Its Computation, pp. 389–440. CSLI Publications, Stanford (2001)
17. Stabler, E.P.: Recognizing head movement. In: de Groote, P., Morrill, G., Retoré, C. (eds.) LACL 2001. LNCS (LNAI), vol. 2099, pp. 245–260. Springer, Heidelberg (2001). doi:10.1007/3-540-48199-0_15
18. Stabler, E.P., Keenan, E.L.: Structural similarity within and among languages. Theoret. Comput. Sci. **29**, 345–363 (2003). Algebraic Methods in Language Processing
19. Titov, I., Henderson, J., Merlo, P., Musillo, G.: Online graph planarisation for synchronous parsing of semantic and syntactic dependencies. In: Proceedings of 21st International Joint Conference on Artifical Intelligence, IJCAI 2009 (2009)
20. Zhang, Y., Clark, S.: Syntactic processing using the generalized perceptron and beam search. Comput. Linguist. **37**(1), 105–151 (2011)