A Simple CF Formalism and Free Word Order

Filip Graliński

Adam Mickiewicz University
Umultowska 87, 61-614 Poznań, Poland
filipg@amu.edu.pl

Abstract

The first objective of this paper is to present a simple grammatical formalism named Tree-generating Binary Grammar. The formalism is weakly equivalent to CFGs, yet it is capable of generating various kinds of syntactic trees, including dependency trees. Its strong equivalence to some other grammatical formalisms is discussed. The second objective is to show how some free word order phenomena in Polish can be captured within the proposed formalism.

1. Introduction

The aim of this paper is twofold. Firstly, a low-level grammatical formalism in binary form (named Tree-generating Binary Grammar – TgBG) is presented. What is important about this formalism is not its weak-generative power (the grammars of the proposed formalism are weakly equivalent to CFGs), but rather its tree-generating capabilities. Secondly, it is shown how free(r) word order phenomena in Polish, including some kinds of discontinuities, can be captured within the proposed formalism.

In Section 2. the formalism itself is presented. Also, we discuss a parsing procedure that gives a shared parse forest and we show how to extract a tree from such a shared forest. In Section 3. it is shown how other formalisms, both dependency and phrase-structure ones, can be converted into TgBGs. We claim that Tree-generating Binary Grammars are some kind of common ground for these formalisms. Section 4. outlines a higher-level interface to TgBGs. In Sections 5. and 6. we show how some specific phenomena of Polish (free order of VP components and some types of discontinuity, respectively) can be described with a Tree-generating Binary Grammar. Final remarks are expressed in Section 7.

2. Tree-generating Binary Grammars

2.1. Preliminary definitions

Definition 1 *A* ***fragment*** *over an alphabet* Σ *is any finite set* $F = \{(w_1, i_1), \ldots, (w_k, i_k)\}$*, where* $w_1, \ldots, w_k \in \Sigma$*,* $i_1, \ldots, i_k \in \mathcal{N}$*,* $i_1 < i_2 < \ldots < i_{k-1} < i_k$*. A fragment* F *is said to be* ***continuous*** *if* $F = \{(w_1, i), (w_2, i+1), \ldots, (w_k, i+k-1)\}$ *for some* $i, k \geq 1$*.*

A fragment can be seen as a set of occurrences of words. It need not be continuous.

We are now going to give a definition of a syntactic tree. Informally, a syntactic tree is a tree whose nodes are labelled with (syntactic) categories or word occurrences. A leaf may be labelled with a category or a word occurrence. Only leaves can be labelled with word occurrences. Tree arcs are labelled with syntactic roles. Typical examples of syntactic roles are: *head*, *object*, *subject*, *modifier*.

Definition 2 *A* ***syntactic tree*** *over an alphabet* Σ*, a set of categories* C [1]*and a set of syntactic roles* R *is a graph* (V, E, θ) *such that:*

- (V, E) *is a rooted tree:* V *is a finite non-empty set of nodes and* $E \subset V \times R \times V$*,*
- $\theta \colon V \to C \cup (\Sigma \times \mathcal{N})$*, where* $\theta(V) \cap (\Sigma \times \mathcal{N})$ *is a fragment;* $\theta(V) \cap (\Sigma \times \mathcal{N})$ *will be called the* ***yield*** *of the tree,*
- $(u, \rho, v) \in E \Rightarrow \theta(u) \in C$ *(internal nodes are labelled with categories),*
- $\theta(v_1) = \theta(v_2) \in \Sigma \times \mathcal{N} \Rightarrow v_1 = v_2$.

The set of all trees over an alphabet Σ*, a set of categories* C *and a set of syntactic roles* R *will be denoted by* $\mathcal{T}(\Sigma, C, R)$*.*

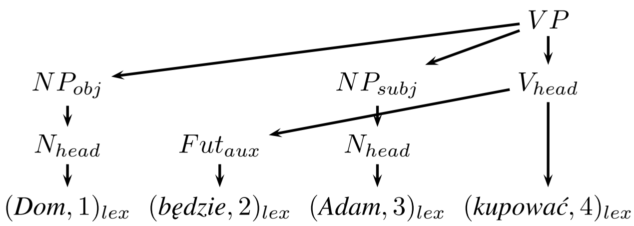

Figure 1: A syntactic tree for sentence (1). Syntactic roles are written as subscripts.

Note that the daughters of a node are not explicitly ordered (formally, a syntactic tree defined above is not an ordered tree). They can, however, be ordered in a natural way if the fragments they cover are continuous.

A sample tree for a Polish sentence (1) is presented in Figure 1. Five categories (VP, NP, V, N, Fut) and five syntactic roles (*obj*, *subj*, *head*, *lex*, *aux*) were used.

(1)	*Dom*	*będzie*	*Adam*	*kupować*
	house(acc)	be(fut)	Adam(nom)	buy(inf)
	'Adam will be buying a house'			

Note that the phrase *będzie… kupować* is discontinuous here.

[1] We assume that $C \cap \Sigma \times \mathcal{N} = \varnothing$.

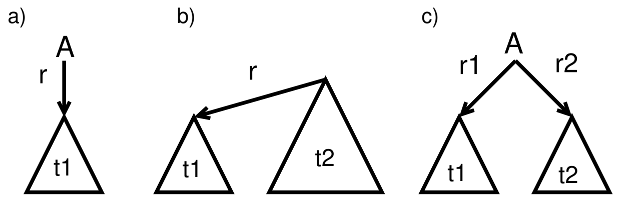

Figure 2: Basic tree operations: (a) unary new node, (b) left attachment, (c) binary new node

Definition 3 *A **unary tree operation** is a unary function $f\colon \mathcal{T}(\Sigma, C, R) \rightarrow \mathcal{T}(\Sigma, C, R) \cup \{\Box\}$. A **binary tree operation** is a function $f\colon \mathcal{T}(\Sigma, C, R) \times \mathcal{T}(\Sigma, C, R) \rightarrow \mathcal{T}(\Sigma, C, R) \cup \{\Box\}$. Special value $\Box$ means that a tree operation is not applicable for given argument(s), in particular it is assumed that $f(t_1, t_2) = \Box$ if f is a binary tree operation and $yield(t_1) \cap yield(t_2) \neq \varnothing$.*

Let us introduce two unary tree operations:

- **identity**, written id: $id(t) := t$,
- **new node** of category $A \in C$ with role $r \in R$, written $nn(A, r)$: $nn(A, r)((V_1, E_1, \theta_1)) := (V, E, \theta)$ where $V = V_1 \cup \{v\}$, $v \notin V_1$, $E = E_1 \cup \{(v, r, root(t))\}$, $\theta = \theta_1 \cup \{(v, A)\}$, see Figure 2a

and three binary tree operations (we assume that $t_1 = (V_1, E_1, \theta_1)$, $t_2 = (V_2, E_2, \theta_2)$, $V_1 \cap V_2 = \varnothing$):

- **left attachment** with role r, written $la(r)$: $la(r)(t_1, t_2) := (V, E, \theta)$ where $V = V_1 \cup V_2$, $E = E_1 \cup E_2 \cup \{(root(t_2), r, root(t_1))\}$, $\theta = \theta_1 \cup \theta_2$, see Figure 2b,
- **right attachment** with role r, written $ra(r)$: $ra(r)(t_1, t_2) := la(r)(t_2, t_1)$,
- **new node** of category $A \in C$ with roles r_1, r_2, written $nn(A, r_1, r_2)$: $nn(A, r_1, r_2) := (V, E, \theta)$ where $V = V_1 \cup V_2 \cup \{v\}$, $v \notin V_1 \cup V_2$, $E = E_1 \cup E_2 \cup \{(v, r_1, root(t_1)), (v, r_2, root(t_2))\}$, $\theta = \theta_1 \cup \theta_2 \cup \{(v, A)\}$, see Figure 2c.

2.2. Main definitions

Definition 4 *Tree-generating Binary Grammar (TgBG) is a tuple $(\Sigma, Q, Q_f, C, R, \mathcal{L}, P)$, where Σ is a finite alphabet, Q is a set of states (variables), $Q_f \subset Q$ is a set of final states, C is a set of categories (for tree nodes), R is a set of syntactic roles, $\mathcal{L}$ is a lexicon, i.e. a set of rules $x \rightarrow w : A : r$, where $x \in Q$, $w \in \Sigma$, $A \in C$, $r \in R$, P is a set of rules, namely:*

- *binary rules: $x \rightarrow y\ z : I$ where $x, y, z \in Q$, I is a binary tree operation,*
- *unary rules: $x \rightarrow y : I$ where $x, y \in Q$, I is a unary tree operation.*

The eight rules of an example TgBG are given below. State names start with a lower-case letter (e.g. n, vp), while categories – with an upper-case letter (e.g. N, VP).

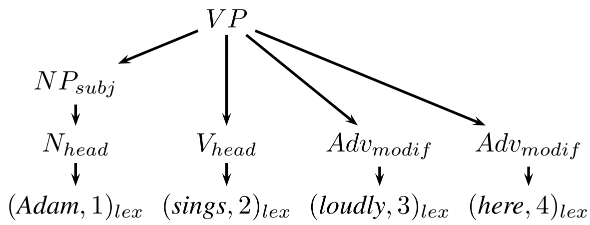

Figure 3: A syntactic tree for grammar the sample grammar

n → Adam *: N : lex*	*np* → *n : nn(NP, head)*
v → sings *: V : lex*	*vp* → *v : nn(VP, head)*
adv → loudly *: Adv : lex*	*vp* → *vp adv : ra(modif)*
adv → here *: Adv : lex*	*s* → *np vp : la(subj)*

Elements of Q are like nonterminals of a standard CFG. The difference is that the trees generated by the grammar are not composed of nonterminals but rather of distinct category symbols (elements of C).

Definition 5 *The **derivability** relation $\vdash_G \ \subset Q \times \mathcal{T}(\Sigma, C, R)$ for grammar G is defined as the smallest relation such that:*

- *$x \vdash_G (\{u, v\}, \{(u, r, v)\}, \theta)$ where $x \in Q$, $\theta(u) = A$, $\theta(v) = (w, i)$ for some lexicon rule $x \rightarrow w : A : r$,*
- *if G has a unary rule $x \rightarrow y : I$ and $y \vdash_G t$, $I(t) \neq \Box$ then $x \vdash_G I(t)$,*
- *if G has a binary rule $x \rightarrow y\ z : I$ and $y \vdash_G t_1$, $z \vdash_G t_2$, $\max(yield(t_1)) = \min(yield(t_2)) - 1$, $I(t_1, t_2) \neq \Box$ then $x \vdash_G I(t_1, t_2)$,*

The function *max* (*min*) returns the position of the rightmost (leftmost) word of a fragment.

For example, in the above sample grammar it is possible to derive the tree depicted in Figure 3 from s.

Definition 6 *The language generated by grammar $G = (\Sigma, Q, Q_f, C, R, \mathcal{L}, P)$, written $L(G)$, is defined as*

$$L(G) = \{w_1 \ldots w_n | x \in Q_f, x \vdash_G t, \quad yield(t) = \{(w_1, 1), \ldots, (w_n, n)\}\}$$

2.3. Parsing algorithm

Standard CKY algorithm (Hopcroft and Ullman, 1979) can be used[2] for an input text of length n to obtain a chart with entries of the form $[i, j, x, L]$ where $0 \leq i < j \leq n$, L is a list of elements of the three possible types:

- $[k, y, z, I]$, where $i < k < j$, $y, z \in Q$, I is a binary tree instruction (an element created with binary rule $x \rightarrow y\ z : I$ and entries $[i, k, y, L_1]$ and $[k, j, z, L_2]$),
- $[y, I]$, where $y \in Q$, I is a unary tree instruction(an element created with unary rule $x \rightarrow y : I$ and entry $[i, j, y, L_1]$),
- t, where t is a tree created with a lexicon rule.

As is the case with the original CKY algorithm, the time complexity is $\mathcal{O}(n^3)$, assuming a fixed grammar.

[2] We assume that there are no chain rule cycles.

2.4. Selecting a tree

With the CKY algorithm described in 2.3. a shared (packed) parse forest is generated. Extracting a tree from a chart is straightforward, see the recursive algorithm for selecting a tree below.

proc select-in-chart(Γ)
    $n :=$ *length of input text*
    select entry $[1, n, x, L]$ *in chart* Γ
    return select-for-entry($[1, n, x, L]$)

proc select-for-entry($[i, j, x, L]$)
    select element e *in* L
    return select-for-element(e,i,j)

proc select-for-element($[k, y, z, I]$,i,j)
    return I(select-for-entry($[i, k, y, L_1]$),
        select-for-entry($[k, j, z, L_2]$))

proc select-for-element($[y, I]$,i,j)
    return I(select-for-entry($[i, j, y, L_1]$))

proc select-for-element(t,i,j)
    return t

There is often no need to extract the whole tree. For example, if the leftmost daughter of the root is to be determined, select-for-entry($[k, j, z, L_2]$) may be omitted.

3. TgBGs and other formalisms

TgBGs are weakly equivalent to CFGs[3] as, without their tree-generating capabilities, they are simply CFGs in binary form. The advantage of TgBGs is that they can simulate other CFG-weakly-equivalent formalisms as to the trees they generate. Generally speaking, with TgBGs it is possible to obtain standard CFG derivation trees as well as dependency trees. Interestingly, a hybrid approach is possible, i.e. getting flat phrase structure trees – flattened versions of full derivation trees. Such flattened trees are much easier to handle in the subsequent processing.

In the present section, only the five tree instructions listed in Section 2.1. are taken into consideration. The repertoire of tree instructions can be extended, e.g. for getting non-projective trees (trees for discontinuous phrases). See more on this in Section 6.

3.1. Standard CFGs

It is not difficult to convert CFG rules into TgBG rules to obtain standard CFG derivation trees as long as rules with an empty right-hand side (RHS) are excluded and terminal symbols occur only in lexical rules (i.e. in rules of the form $A \to w$).[4] As an example consider a rule $A \to B\ C\ D\ E$, which can be converted into the following TgBG binary rules:

- $a_0 \to b\ c : nn(sub, sub)$
- $a_1 \to a_0\ d : ra(sub)$
- $a \to a_1\ e : ra(sub)$

We assume that b, c, d, e will be used in analogous TgBG rules created for the CFG rules with respectively B, C, D, E on the left-hand sides (LHSs). As syntactic roles are irrelevant here, we simply put a dummy role (sub) in each rule.

3.2. Free-Order Dependency Grammars

Free-Order Dependency Grammar (FODG) is a grammatical formalism intended for languages with a high degree of word order freedom (such as Czech) (Holan et al., 1998). In some respects TgBGs are similar to FODGs. The main difference is that FODG is a pure dependency formalism, whereas TgBGs can embrace both dependency and phrase structure approach. Also, by default, fragments covered by symbols of the RHS of a rule need not be adjacent (nor continuous) in a FODG.[5] We assume here a fixed-order interpretation of FODG rules. Naturally, TgBG rules can be reinterpreted in a FODG-like manner to account for discontinuities. An alternative approach to discontinuity within TgBGs will be discussed in Section 6.

There are four types of rules in FODGs: $A \to w$, $A \to B$, $A \to_L B\ C$, $A \to_R B\ C$ where A, B, C are nonterminal symbols and w is a terminal symbol.[6] These rules can be mapped into TgBG rules respectively:

- $a \to w : A : lex$
- $a \to b : id$
- $a \to b\ c : ra(sub)$
- $a \to b\ c : la(sub)$

This way, trees corresponding to FODG dependency trees (Dϵ-trees) will be created. A small difference would be that all the nodes of a Dϵ-tree are labelled with words, whereas the internal nodes of the corresponding TgBG tree will be labelled with category symbols.

3.3. Generative Dependency Grammars

Generative Dependency Grammars (GDGs) (Nasr and Rambow, 2004) are lexicalized Extended Context-Free Grammars.[7] This dependency formalism can be simulated with TgBGs as well: a GDG rewriting rule should be broken into binary TgBG rules in such a way that the symbols to the right of the head are considered first and then, when the right part is completed, the symbols on the left are incorporated. For example, GDG rule $V \to N\ \textit{saw}\ N\ P*$ can be rendered with the following TgBG rules:

- $v_0 \to \textit{saw} : V : lex$
- $v_1 \to v_0\ n : ra(obj)$

[3]As regards languages without the empty string.

[4]It is possible to remove these limitations by extending the set of tree instructions.

[5]This freedom can be limited with rule modifiers.

[6]In fact, terminal symbols can occur in binary rules, though the case will be disregarded here. It can, however, be captured with an extended set of TgBG tree instructions.

[7]Extended CFGs are like CFGs, the only difference is that RHSs are regular expressions over the symbols of the grammar.

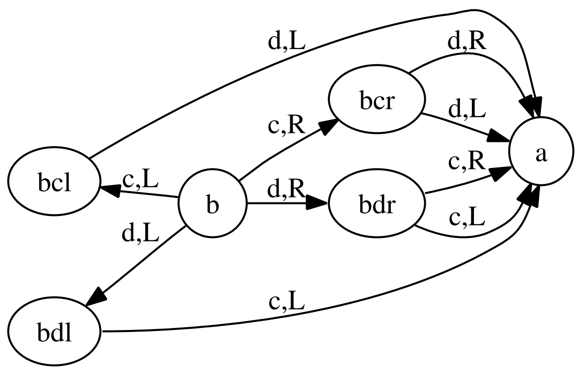

Figure 4: TgBG rules simulating poms-rule $A \rightarrow B \parallel C \parallel D$. An arc labelled z, R (resp. z, L) connecting nodes y and x represents a rule $x \rightarrow y\ z : ra(sub)$ (resp. $x \rightarrow z\ y : la(sub)$).

- $v_1 \rightarrow v_1\ p : ra(modif)$
- $v \rightarrow n\ v_1 : la(subj)$

3.4. Partially ordered multiset CFGs

Partially ordered multiset CFGs (poms-CFGs) (Nederhof et al., 2003) are an extension of CFGs in which expressions with an interleaving operator ($\parallel$) may be used as RHSs. [8] As poms-CFGs are strongly equivalent to CFGs, it is trivially possible to simulate them with TgBG rules. What is of interest is whether TgBG rules can be specified in such a way that parsing become as effective as with poms-CFGs. This goal can be achieved by choosing a "head" in a poms-CFG rule[9] and then proceeding similarly as in the case of GDGs (see 3.3.). For example, consider poms-CFG rule $A \rightarrow B \parallel C \parallel D$ (each permutation of BCD is allowed). With B selected as the head, the set of TgBG rules depicted as the "automaton" in Figure 4 would be equivalent to this rule.

Note that as few variables were introduced as there are states in the poms-automaton for poms-expression $B \parallel C \parallel D$. The spurious ambiguities resulting from attaching the left constituent first were avoided.

4. Higher-level interface to TgBGs

Some enhancements to TgBGs should be made to facilitate creating real grammars. Here we discuss introducing attributes – the enhancement that is particularly useful for languages with rich inflectional system (such as Polish) – i.e. states may be augmented with attributes and TgBG rules may contain attribute expressions. Attribute expressions are used to check conditions (such as agreement conditions) and to set values for attributes of new nodes. It should be emphasised that rules with attribute expressions can be compiled into rules with atomic states, assuming the set of attributes and their values is finite.

Standard C operators (assignment =, equality ==, negation `!`, AND operator `&&`, OR operator `||`) are used in attribute expressions. Attribute names start with an upper-case letter. Attributes of existing nodes are referenced with a dot (e.g $ap.Case$), while attributes of the newly created node are referred to in a direct manner (e.g. $Case$). By default, attributes of the new node are first copied from the "head" marked with @. Attribute expressions are written after a colon. Here is a sample rule for noun phrases (incorporation of an adjective phrase).

(2) *np → ap np@ : la(modif) : ap.Case == Case*
&& ap.Num == Num && ap.Gender == Gender

A macro preprocessor would be another useful element of a higher-level interface to TgBGs (see remarks in Section 5.).

5. TgBG and free order in VPs

In Polish the order of VP components is relatively free. For example each of the 24 permutations of the sentence

(3) *Adam* *mocno* *kocha* *Barbarę*
Adam(nom) deeply love(3.sg) Barbara(acc)
'Adam loves Barbara deeply'

seems to be grammatical, although some of them are more and some less frequent or natural (the standard order is SVO, as in (3)). The word order is influenced by the thematic-rhematic structure of the utterance; however, here we abstract from such phenomena.

In order to capture the free order in VPs we propose the following skeleton of TgBG rules (cf. the rules presented in Figure 4):

(4) *rvp → v@ : nn(VP, head)*

First the right constituents are attached:

(5) *rvp → rvp@ advp : ra(modif)*

(*advp* stands for *adverbial phrase*.)

(6) *rvp → rvp@ np : ra(obj) : np.Case == ObjCase*
&& !ObjFilled && ObjFilled = 1

(*ObjCase* is the case required for the object, e.g. *acc* = *accusative* for *kochać* (*love*).)

(7) *rvp → rvp@ np : ra(subj) : np.Case == nom*
&& np.Gender==Gender
&& np.Num==Num
&& np.Person==Person
&& !SubjFilled && SubjFilled = 1

(*SubjFilled* is used to ensure that at most one subject is attached.)
We switch to the left side:

[8] Interestingly, operator * was not allowed in the original proposal of poms-CFGs. If it had been, poms-CFGs would have greater strong generative power, just as Extended CFGs have.

[9] This may involve splitting a poms-CFG rule into several rules. In real cases, however, it can be avoided as it seems.

(8) *lvp → rvp@ : id*

(9) *lvp → advp lvp@ : la(modif)*

(10) *lvp → np lvp@ : la(obj) : np.Case == ObjCase*
&& !ObjFilled && ObjFilled = 1

(11) *lvp → np lvp@ : la(subj) : np.Case == nom*
&& np.Gender==Gender
&& np.Num==Num
&& np.Person==Person
&& !SubjFilled && SubjFilled = 1

(12) *vp → lvp@ : id*

Now the VP is completed.

(13) *s → vp : nn(S, head)*

Some of the above rules are dual: a rule for attaching a constituent on the right and an analogous rule for attaching on the left ((5) and (9), (6) and (10), (7) and (11)). Such rules can be written in a more concise manner with a macro preprocessor (the shared part of the rules, e.g. agreement conditions, would be specified only once).

It is possible to account for limitations in free word order within this approach. For example, if we assume that an infinitive subphrase closes the verb phrase on the right (i.e. no other constituents can follow it)[10] the following rule could be specified:

(14) *lvp → rvp@ vp : ra(inf) : InfExpected*
&& vp.Tense == inf

where *InfExpected* attribute is set for such verbs as *chcieć* (*want*), *lubić* (*like*), etc.

6. TgBG and discontinuity

We are now going to show how at least some types of discontinuity can be handled in a TgBG.

A new binary tree instruction **left insert** (li) is introduced. The result of $li(q, r)(t_1, t_2)$ is t_2 with t_1 attached to that daughter of $root(t_2)$ whose arc is labelled with role q (the result is $\square$ if there are zero or more than one such daughters), see Figure 5. Let us also introduce the symmetrical tree instruction, **right insert**, written $ri(q, r)$: $ri(q, r)(t_1, t_2) := li(q, r)(t_2, t_1)$.

Although the set of tree instructions is extended, the $\mathcal{O}(n^3)$ parsing algorithm (see Section 2.3.) remains the same. In general a chart may contain spurious entries as li/ri may return $\square$ (such entries should be pruned). Note that the sample rules given in the present paper are written in such a way that no li/ri will return $\square$.

6.1. Polish analytical future form

Let us consider Polish analytical future form, which can be discontinuous (like *będzie… kupować* in sentence (1)). It is assumed that a tree like in Figure 1 is expected. First we focus on the case where *będzie* (or any future form of verb *być*) precedes the infinitive (like in sentence (1)).

[10]In practice this seems to be a reasonable approximation.

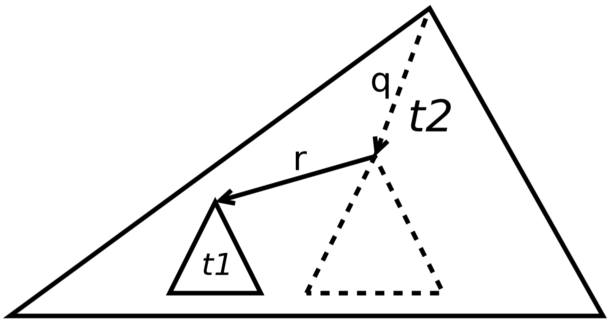

Figure 5: Left insert

(15) *lvp → be lvp@ : li(head,aux) :*
(Tense == inf && Person = be.Person
Gender = be.Gender && Num = be.Num
||
Tense == fut_wanted && be.Person == SubjP
be.Gender == SubjG && be.Num == SubjN)
&& Tense = fut

(Variable *be* represent a future form of verb *być* (*to be*), e.g. *będzie*. The second part of the alternative applies if the subject has already been attached.)

(16) *lvp → np lvp@ : la(subj) :*
Tense == inf && np.Case == nom
&& Tense = fut_wanted && SubjP = np.Person
&& SubjG = np.Gender && SubjN = np.Num

(A subject is tentatively attached to an infinitive.)

The symmetrical cases, in which the infinitive precedes the form of *być* (*kupować… będzie*), can be captured with the following rules:

(17) *rvp → rvp@ be : ri(head,aux) :*
(Tense == inf && Person = be.Person
Gender = be.Gender && Num = be.Num
||
Tense == fut_wanted && be.Person == SubjP
be.Gender == SubjG && be.Num == SubjN)
&& Tense = fut

(18) *rvp → rvp@ np : ra(subj) :*
Tense == inf && np.Case == nom
&& Tense = fut_wanted && SubjP = np.Person
&& SubjG = np.Gender && SubjN = np.Num

Rules (17) and (18) enable handling sentences like *Kupować Adam będzie dom.* In practice, rules (17) and (18) are similar enough to rules, respectively, (15) and (16) to be generated from one macro.

Finally, rule (12) should be modified to avoid treating phrases like *Adam kupować* as normal VPs:

(19) *vp → lvp@ : id : Tense != fut_wanted*

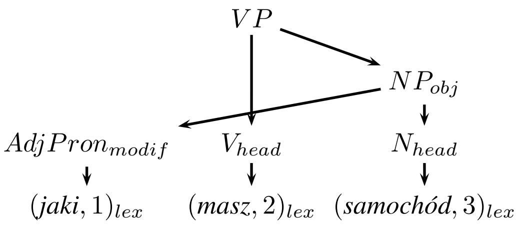

Figure 6: A syntactic tree for sentence (21).

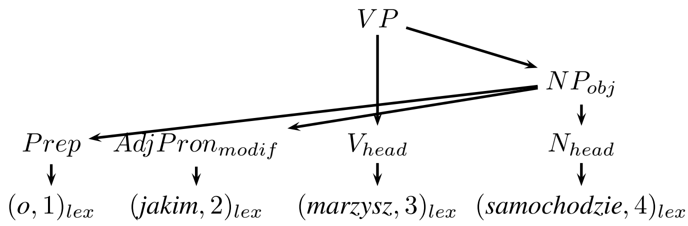

Figure 7: A syntactic tree for sentence (24).

6.2. Adjectival interrogative pronouns

In Polish, interrogative pronouns[11] should open the clause (except if a preposition occurred, then the pronoun should be preceded with the preposition). Interrogative pronouns often introduce discontinuity. In fact, there are cases of sentences with such pronouns where discontinuity is more natural than continuity. For example, sequence (20) is more frequent[12] than a synonymous sequence without a discontinuity (21).

(20) *jaki* *masz* *samochód* *?*
what(acc) have(2.sg) car(acc)
'what car do you have?'

(21) *jaki* *samochód* *masz* *?*
what(acc) car(acc) have(2.sg)
'what car do you have?'

Let us examine how this kind of discontinuity (like the one in sentence (20)) could be captured within a TgBG. First, rule (6) should be modified:

(22) *rvp → rvp@ np : ra(obj) : np.Case == ObjCase*
&& !ObjFilled && ObjFilled = 1
&& ObjN = np.Num
&& ObjG = np.Gender

Then let us introduce a new rule for adjectival interrogative pronouns used as an element of an object NP:

(23) *vp → adj_ipron lvp@ : li(obj,modif) :*
ObjFilled && adj_ipron.Case == ObjCase
&& adj_ipron.Num == ObjN
&& adj_ipron.Gender == ObjG

We check whether an object NP was included in the VP and whether number/gender agreement is fulfilled. For sentence (20) the tree depicted in Figure 6 is generated.

The case of discontinuous PPs is more difficult than that of discontinuous NPs. Consider, for example, sentence:

(24) *O* *jakim* *marzysz* *samochodzie?*
about what(loc) dream(2.sg) car(loc)
'What car are you dreaming about?'

[11] *jaki* (*what*), *który* (*which*), *czyj* (*whose*)

[12] 523 : 28 according to Google search engine (September 2005).

Note that the string *marzysz samochodzie*, in contrast to *masz samochód* in (20), do not constitute a VP. This means that modifying the rule for attaching PP to a VP[13] will not suffice and a new rule for a noun phrase being a part of discontinuous PP should be introduced. This rule is as follows:

(25) *rvp → rvp@ np : ra(discontpp) :*
&& !DiscontPP && DiscontPP = 1
&& DiscontPPC = np.Case
&& DiscontPPN = np.Num
&& DiscontPPG = np.Gender

Such pseudo-VPs (as *marzysz samochodzie*) can be turned into a VP with the following two rules:

(26) *lvp_dpp → adj_ipron lvp@ : li(discontpp, modif) :*
DiscontPP
&& adj_ipron.Case == DiscontPPC
&& adj_ipron.Num == DiscontPPN
&& adj_ipron.Gender == DiscontPPG

(*lvp_dpp* is an auxiliary variable.)

(27) *vp → prep lvp_dpp@ : li(discontpp, prep) :*
DiscontPP
&& prep.Case == DiscontPPC

This way the tree depicted in Figure 7 can be generated. Note that a PP node is not introduced; however, this could be accomplished if compositions of basic tree operations could be specified as tree operations within TgBG rules.

Finally, rule (19) should be further modified so that pseudo-VPs like *marzysz samochodzie* are not treated as normal VPs:

(28) *vp → lvp@ : id :*
Tense != fut_wanted
&& !DiscontPP

7. Conclusion and future work

A formalism which is weakly equivalent to CFGs yet capable of describing various kinds of trees was presented. Within this formalism it is possible to capture some types of free word order.

Future work will focus on delimiting the extent of discontinuity in Polish that can successfully be handled in TgBG framework and on determining minimal enhancements required to embrace the remaining part.

The important problem that still remains is how to capture verb chains and their interaction with various types of discontinuity.

[13] A rule $rvp \rightarrow rvp@\ \ pp : ra(modif)$ or similar.

8. References

Holan, Tomáš, Vladislav Kuboň, Karel Oliva, and Martin Plátek, 1998. Two useful measures of word order complexity. In *Proceedings of the COLING-98 Workshop "Processing of Dependency-Based Grammars"*.

Hopcroft, Jeffrey D. and John E. Ullman, 1979. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley.

Nasr, Alexis and Owen Rambow, 2004. A simple string-rewriting formalism for dependency grammar. In Geert-Jan M. Kruijff and Denys Duchier (eds.), *COLING 2004 Recent Advances in Dependency Grammar*.

Nederhof, Mark-Jan, Giorgio Satta, and Stuart M. Shieber, 2003. Partially ordered multiset context-free grammars and free-word-order parsing. In *Proceedings of the 8th International Workshop on Parsing Technologies*.