Conversions between Midly Context Sensitive Grammars

Matthieu Guillaumin

Abstract

This document is a report for my internship for the first year at the MMFAI. The goal of this internship was to implement an efficient translation between grammars belonging to the same complexity class, but involving different formalisms and perspectives, in order to use for all of them the same state-of-the-art parser. The conversion algorithms will be presented as well as the implementations in Ocaml. Finally, we will show with chosen examples the benefits of these conversions for faster parsing.

Contents

1 The place of midly context sensitive languages in language complexity

In the attempt to capture natural language structure into a computational formalism, it is of primary importance to evaluate natural language complexity. Noam Chomsky has proposed in 1956 a containment hierarchy of classes of formal grammars that generate formal languages. The Chomsky hierarchy consists in the following levels:

- Type-0 grammars (unrestricted grammars) include all formal grammars. They generate exactly all languages that can be recognized by a Turing machine. The language that is recognized by a Turing machine is defined as all the strings on which it halts. These languages are also known as the recursively enumerable languages. Note that this is different from the recursive languages which can be decided by an always halting Turing machine.
- Type-1 grammars (context-sensitive grammars) generate the context-sensitive languages. These grammars have rules of the form $\alpha A\beta \rightarrow \alpha\gamma\beta$ with A a nonterminal and α, β and γ strings of terminals and nonterminals. The strings α and β may be empty, but γ must be nonempty. The rule $S \rightarrow \varepsilon$ is allowed if S does not appear on the right side of any rule. The languages described by these grammars are exactly all languages that can be recognized by a non-deterministic Turing machine whose tape is bounded by a constant times the length of the input.
- Type-2 grammars (context-free grammars) generate the context-free languages. These are defined by rules of the form $A \rightarrow \gamma$ with A a nonterminal and γ a string of terminals and nonterminals. These languages are exactly all languages that can be recognized by a non-deterministic pushdown automaton.
- Type-3 grammars (regular grammars) generate the regular languages. These languages are exactly all languages that can be decided by a finite state automaton.

As explained in [10] and [4], natural languages are strictly more expressive than context-free languages, and context-sensitive languages do capture them. On the other hand, context-sensitive languages include languages like $\{a^{2^n} | n \in \mathbb{N}\}$, which is a scheme never occuring in natural language. With the idea that the more general parsing is, the slower, many other formalisms properly included between context-free and context-sensitive languages have been proposed, almost all of them being part of a class introduced by Joshi in 1987 and also in [5] under the name of "midly context sensitive languages". This class includes languages derived from grammars like multiple context free grammars (MCFG, introduced by Seki *et al.* in [7]), minimalist grammars (MG, Stabler's formalization [8] of Chomsky's minimalist program [2]), tree adjoining grammars (TAG, Joshi), and others.

2 Different weak-equivalent forms of MCSG

2.1 Multiple Context-Free Grammars (MCFG)

As introduced in Seki *et al.*, MCFGs are defined as a particular case of *generalized multiple context-free grammar* (form Pollard, 1984). Let's give a direct definition of a m-MCFG: A 5-tuple $G = (N, O, F, P, S)$ is a m-MCFG if

1. N is a finite set of nonterminal symbols.
2. O is a set of n-tuples of strings ($n \geq 1$) over an alphabet T disjoint with N.
3. F is a finite set of partial functions from finite dimensional direct products O^q to O (defining F as a $\bigcup_{q\in\mathbb{N}} F_q$).

4. P is a finite subset of $\bigcup_q (F_q \times N^{q+1})$. An element of P is called a rule and if $(f, A_0, A_1, \ldots, A_q) \in P$ then we write $A_0 \to f(A_1, \ldots, A_q)$.
5. $S \in N$ is the initial symbol.

Terminal rules are those where $q = 0$, i.e where $f \in O$. The following conditions must also be met

1. $O = \bigcup_{i=1}^{m} (T^*)^i$.
2. If $a(f)$ is the arity of $f \in F$, then for each $f \in F$, positive integers $r(f)$ and $d_i(f)$ (for $1 \leq i \leq a(f)$) are given and f is a function from $(T^*)^{d_1(f)} \times \ldots \times (T^*)^{d_{a(f)}(f)}$ to $(T^*)^{r(f)}$.
3. if f^h is the h-th component of f, then f^h is a concatenation of some string constant in T^* and some variables in $X = \{x_{ij} | 1 \leq i \leq a(f), 1 \leq j \leq d_i(f)\}$ in an order defined by f^h. (More formally, if $\overline{x_i} = (x_{i1}, \ldots, x_{id_i(f)})$ then $f^h(\overline{x_1}, \ldots, \overline{x_{a(f)}}) = \alpha_{h0} z_{h1} \alpha_{h1} \ldots z_{hv_h(f)} \alpha_{hv_h(f)}$ where $\alpha_{hk} \in T^*$ and $z_{hk} \in X$).
4. A positive integer $d(A)$ is given to every nonterminal symbol $A \in N$. If $A_0 \to f(A_1, \ldots, A_q)$ is a rule, then $r(f) = d(A_0)$ and $d_i(f) = d(A_i)$ for $1 \leq i \leq a(f)$.
5. For the final symbol S, $d(S) = 1$
6. For each h $(1 \leq h \leq r(f))$ and each variable $x_{ij} \in X$, the total number of occurences of x_{ij} in the right-hand side of f^h is at most one (meaning that there is no copy).

Seki *et al.* also proves in [7] the following results about MCFG:

Theorem 2.1 $CFL = 1 - MCFL \subsetneq 2 - MCFL \subsetneq \ldots MCFL \subsetneq CSL$

Theorem 2.2 $CLF = 1 - MCFG \subsetneq TAL \subset 2 - MCFG$

Lemma 2.3 *For any* m, $\{a^{2^n} | n \in \mathbb{N}\}$ *does not belong to* $m - MCFL$

The parser (by Dan Albro, UCLA, 2000, presented in [1]) we are willing to use is limited to a "binary normal form" of m-MCFGs:

- $F = \bigcup_{q=0}^{2}$, so $a(f) \leq 2$
- The x_{hk} variables are used exactly once in f
- There are no constant strings α_{hk}

They are therefore 4 types of rules in this system:

1. Terminating: `T ->` <*string*>
2. Empty: `T ->` "''
3. Chain: `NT -> NT1` <*map*>
4. Binary: `NT -> NT1 NT2` <*map*>

Where <*map*> represents the function f as a sequence of $a(f)$ sequences of coordinates of components of `NT1` and `NT2`. For example, if `NT1` is of arity 4, `NT2` is of arity 5 and `NT` of arity 3 (note that the constraints entails that this arity is between 1 and 9), the following rule is valid:

```
NT -> NT1 NT2 [0,1;1,1;1,4;0,3][1,0;0,0;1,2][0,2;1,3]
```

Meaning that if (a, b, c, d) is a string matched by `NT1` and (e, f, g, h, i) by `NT2` then $(bfid, eag, ch)$ is recognized as a `NT`.

Our efforts is section 3 will be aimed at this system.

2.2 Minimalist Grammars (MG)

Minimalist grammars (Stabler, 1997, [8]) are inspired by the minimalist program by Chomsky (1995; [2]) and is one of the simplest approach to natural language formalisation. For linguists, it has the advantage to make it easy to write rules close to X-bar theory and keep interesting properties like movements. Given a finite set of symbols B and a set of starting categories $I \subset B$, a 5-tuple $G = (\Sigma, F, T, L, \mathcal{F})$ is a MG if:

1. Σ is a nonempty alphabet
2. A set F of features
$$\begin{array}{rcll} F & = & B & \text{(a nonempty set of basic features)} \\ & & \cup\{= f | f \in B\} & \text{(selection features)} \\ & & \cup\{+f | f \in B\} & \text{(licensor features)} \\ & & \cup\{-f | f \in B\} & \text{(licensee features)} \end{array}$$
3. $T = \{::, :\}$ (types: lexical, derived)
4. $L \subset (\Sigma^* \times \{::\} \times F^*)$ is the lexicon
   If we define the set E of expressions (nonempty sequences of chains) with $E = (\Sigma^* \times T \times F^*)^+$ then
5. $\mathcal{F} = \{\text{merge}, \text{move}\}$ is a set of generating functions, which are partial functions from E^* to E
6. The corresponding language is $L(G) = \text{closure}(L, \mathcal{F})$, and we look only at the strings of category in I: $S_I(G) = \{s | s \cdot f \in L(G), \quad f \in I, \cdot \in T\}$

We still have to define the merge and move functions.

The merge function: merge : $E \times E \to E$ is the union of the following 3 functions, for s, $t \in \Sigma^*$, for $\cdot \in T$, for $f \in B$, $\gamma \in F^*$, $\delta \in F^+$ and for chains $\alpha_1, \ldots, \alpha_k, \beta_1, \ldots, \beta_l$ $(0 \le k, l)$

$$\frac{s ::= f\gamma \qquad t \cdot f, \alpha_1, \ldots, \alpha_k}{st : \gamma, \alpha_1, \ldots, \alpha_k} : \text{merge}_1 \quad \text{(lexical item selects a non-mover)}$$

$$\frac{s := f\gamma, \alpha_1, \ldots, \alpha_k \qquad t \cdot f, \beta_1, \ldots, \beta_l}{ts : \gamma, \alpha_1, \ldots, \alpha_k, \beta_1, \ldots, \beta_l} : \text{merge}_2 \quad \text{(derived item selects a non-mover)}$$

$$\frac{s \cdot = f\gamma, \alpha_1, \ldots, \alpha_k \qquad t \cdot f\delta, \beta_1, \ldots, \beta_l}{s : \gamma, \alpha_1, \ldots, \alpha_k, t : \delta, \beta_1, \ldots, \beta_l} : \text{merge}_3 \quad \text{(any item selects a mover)}$$

The move function: move : $E \to E$ is the union of the following 2 functions, for s, $t \in \Sigma^*$, for $f \in B$, $\gamma \in F^*$, $\delta \in F^+$ and for chains $\alpha_1, \ldots, \alpha_k$ $(0 \le k)$ satisfying the condition: $\exists i, \alpha_i$ has $-f$ as its first feature and none of α_j for $i, j \in [1, k], j \neq i$ has $-f$ as its first feature,

$$\frac{s : +f\gamma, \alpha_1, \ldots, \alpha_{i-1}, t : -f, \alpha_{i+1}, \ldots, \alpha_k}{ts : \gamma, \alpha_1, \ldots, \alpha_{i-1}, \alpha_{i+1}, \ldots, \alpha_k} : \text{move}_1 \quad \text{(final move of licensee phrase)}$$

$$\frac{s : +f\gamma, \alpha_1, \ldots, \alpha_{i-1}, t : -f\delta, \alpha_{i+1}, \ldots, \alpha_k}{s : \gamma, \alpha_1, \ldots, \alpha_{i-1}, t : \delta, \alpha_{i+1}, \ldots, \alpha_k} : \text{move}_2 \quad \text{(nonfinal move of licensee phrase)}$$

As proved combining results from [3] and [6]:

Theorem 2.4 *: Minimalist Grammars and Multiple Context-Free Grammars are weakly equivalent*

We are only going to use the fact that from each MG, we can find a MCFG recognizing exactly the same strings, but both ways are true.

2.3 Minimalist Grammars with head movement, affix hopping and adjunction (hMG)

This definition of minimalist grammars is unfortunetaly not pratical when linguists want to add to these grammars such observations as “head movement”, “affix hopping” and “adjunction”, which happen for example respectively in inversion of subject, presence of “-s” at the third person in non auxiliary verbs, and adding prepositional phrases to a sentence (more details for head movement and affix hopping in [9]). The MGs above can be extended all these sorts of movements while keeping a concise grammar. As the “head movement” suggests, we have to keep several separated strings for a unique category. The head lies in the middle of the string, we split it in three: left of the head (specifier), the head, right of the head (complement).

Instead of expression of the form $s_1 \cdot \gamma_1, s_2 \cdot \gamma_2, \ldots, s_n \cdot \gamma_n$, we will now handle $s, h, c \cdot \gamma_1, s_2 \cdot \gamma_2, \ldots, s_n \cdot \gamma_n$. We have to modify all the above definitions of MG, and in particular the lexicon L, with elements containing now a triple of string but only the head can be nonempty. We also have to add elements to F, in order to trigger head movement: let $R = \{<= f | f \in B\}$ be the *right-incorporators* and $L = \{=> f | f \in B\}$ be the *left-incorporators*. For affix hopping, we have to add two new kind of features, still to be able to trigger the movement: $R' = \{<== f\}$ and $L' = \{==> f\}$ for any $f \in B$ will be respectively *left* and *right* affix hopping, this is somehow the inverse operation of head movement. F is now $F = B \cup S \cup M \cup N \cup R \cup L \cup R' \cup L'$. All this implies to redefine the merge and move function, adding more cases.

The merge function is now the union of 11 functions:

$$\frac{\varepsilon, s, \varepsilon ::= f\gamma \qquad t_s, t_h, t_c \cdot f, \alpha_1, \ldots, \alpha_k}{\varepsilon, s, t_s t_h t_c : \gamma, \alpha_1, \ldots, \alpha_k} : \text{r1'}$$

$$\frac{\varepsilon, s, \varepsilon ::<= f\gamma \qquad t_s, t_h, t_c \cdot f, \alpha_1, \ldots, \alpha_k}{\varepsilon, s t_h, t_s t_c : \gamma, \alpha_1, \ldots, \alpha_k} : \text{r1right}$$

$$\frac{\varepsilon, s, \varepsilon ::=> f\gamma \qquad t_s, t_h, t_c \cdot f, \alpha_1, \ldots, \alpha_k}{\varepsilon, t_h s, t_s t_c : \gamma, \alpha_1, \ldots, \alpha_k} : \text{r1left}$$

$$\frac{s_s, s_h, s_c := f\gamma, \alpha_1, \ldots, \alpha_k \qquad t_s, t_h, t_c \cdot f, \beta_1, \ldots, \beta_l}{t_s t_h t_c s_s, s_h, s_c : \gamma, \alpha_1, \ldots, \alpha_k, \beta_1, \ldots, \beta_l} : \text{r2'}$$

$$\frac{s_s, s_h, s_c \cdot = f\gamma, \alpha_1, \ldots, \alpha_k \qquad t_s, t_h, t_c \cdot f\delta, \beta_1, \ldots, \beta_l}{s_s, s_h, s_c : \gamma, \alpha_1, \ldots, \alpha_k, t_s t_h t_c : \delta, \beta_1, \ldots, \beta_l} : \text{r3'}$$

$$\frac{s_s, s_h, s_c \cdot <= f\gamma, \alpha_1, \ldots, \alpha_k \qquad t_s, t_h, t_c \cdot f\delta, \beta_1, \ldots, \beta_l}{s_s, s_h t_h, s_c : \gamma, \alpha_1, \ldots, \alpha_k, t_s t_c : \delta, \beta_1, \ldots, \beta_l} : \text{r3right}$$

$$\frac{s_s, s_h, s_c \cdot <= f\gamma, \alpha_1, \ldots, \alpha_k \qquad t_s, t_h, t_c \cdot f\delta, \beta_1, \ldots, \beta_l}{s_s, t_h s_h, s_c : \gamma, \alpha_1, \ldots, \alpha_k, t_s t_c : \delta, \beta_1, \ldots, \beta_l} : \text{r3left}$$

$$\frac{\varepsilon, s, \varepsilon ::==> f\gamma \qquad t_s, t_h, t_c \cdot f, \alpha_1, \ldots, \alpha_k}{\varepsilon, \varepsilon, t_s t_h s t_c : \gamma, \alpha_1, \ldots, \alpha_k} : \text{r1hopright}$$

$$\frac{\varepsilon, s, \varepsilon ::<== f\gamma \qquad t_s, t_h, t_c \cdot f, \alpha_1, \ldots, \alpha_k}{\varepsilon, \varepsilon, t_s s t_h t_c : \gamma, \alpha_1, \ldots, \alpha_k} : \text{r1hopleft}$$

$$\frac{\varepsilon, s, \varepsilon ::==> f\gamma, \alpha_1, \ldots, \alpha_k \qquad t_s, t_h, t_c \cdot f\delta, \beta_1, \ldots, \beta_l}{\varepsilon, \varepsilon, \varepsilon : \gamma, \alpha_1, \ldots, \alpha_k, t_s t_h s t_c : \delta, \beta_1, \ldots, \beta_l} : \text{r3hopright}$$

$$\frac{\varepsilon, s, \varepsilon ::<== f\gamma, \alpha_1, \ldots, \alpha_k \qquad t_s, t_h, t_c \cdot f\delta, \beta_1, \ldots, \beta_l}{\varepsilon, \varepsilon, \varepsilon : \gamma, \alpha_1, \ldots, \alpha_k, t_s s t_h t_c : \delta, \beta_1, \ldots, \beta_l} : \text{r3hopleft}$$

The move function changes only in a trivial way:

$$\frac{s_s, s_h, s_c : +f\gamma, \alpha 1, \ldots, \alpha_{i-1}, t : -f, \alpha_{i+1}, \ldots, \alpha_k}{ts_s, s_h, s_c : \gamma, \alpha 1, \ldots, \alpha_{i-1}, \alpha_{i+1}, \ldots, \alpha_k} : \text{m1'}$$

$$\frac{s_s, s_h, s_c : +f\gamma, \alpha 1, \ldots, \alpha_{i-1}, t : -f\delta, \alpha_{i+1}, \ldots, \alpha_k}{s_s, s_h, s_c : \gamma, \alpha 1, \ldots, \alpha_{i-1}, t : \delta, \alpha_{i+1}, \ldots, \alpha_k} : \text{m2'}$$

Adding adjunction requires to add two other sets to the grammar. There are two types of adjunction: *left* and *right*. Each of them is a relation between categories or more generally between chains beginning with a category. Such a relation is noted with $\ll$ (for right adjunction) and $\gg$ (for left). For example: $a \gg N$ or $D - k \ll D - k$. We have to add a new function to our $\mathcal{F}$ set: adjoin, which is the union of the following 4 functions:

$$\frac{s_s, s_h, s_c \cdot f\gamma, \alpha_1, \ldots, \alpha_k \qquad t_s, t_h, t_c \cdot g\eta\nu, \beta_1, \ldots, \beta_l}{s_s s_h s_c t_s, t_h, t_c : g\eta\nu, \alpha_1, \ldots, \alpha_k, \beta_1, \ldots, \beta_l} : \text{left-adjoin1 if } f\gamma \gg g\eta$$

$$\frac{s_s, s_h, s_c \cdot f\gamma, \alpha_1, \ldots, \alpha_k \qquad t_s, t_h, t_c \cdot g\eta\nu, \beta_1, \ldots, \beta_l}{t_s, t_h, t_c s_s s_h s_c : g\eta\nu, \alpha_1, \ldots, \alpha_k, \beta_1, \ldots, \beta_l} : \text{right-adjoin1 if } g\eta \ll f\gamma$$

$$\frac{s_s, s_h, s_c \cdot f\gamma\delta, \alpha_1, \ldots, \alpha_k \qquad t_s, t_h, t_c \cdot g\eta\nu, \beta_1, \ldots, \beta_l}{t_s, t_h, t_c : g\eta\nu, s_s s_h s_c : \delta, \alpha_1, \ldots, \alpha_k, \beta_1, \ldots, \beta_l} : \text{left-adjoin2 if } f\gamma \gg g\eta$$

$$\frac{s_s, s_h, s_c \cdot f\gamma\delta, \alpha_1, \ldots, \alpha_k \qquad t_s, t_h, t_c \cdot g\eta\nu, \beta_1, \ldots, \beta_l}{t_s, t_h, t_c : g\eta\nu, s_s s_h s_c : \delta, \alpha_1, \ldots, \alpha_k, \beta_1, \ldots, \beta_l} : \text{right-adjoin2 if } g\eta \ll f\gamma$$

The resulting hMGs are weakly-equivalent to MGs, as Michaelis showed in [6], so it is also to MCFG.

2.4 minimalist Tupled Pregroup Grammars (mTPG)

Stabler proposed in [11] an extension to Lambek's Pregroup Grammars called minimalist Tupled Pregroup Grammar: $G = \langle \Sigma, \mathbb{P}, \leq, \mathbb{I}, S \rangle$ is a mTPG if:

1. Σ is a nonempty alphabet
2. $\mathbb{P}$ is a set of simple types partially ordered by $\leq$
3. $\mathbb{I}$ is a subset of $(\mathbb{T} \times (\Sigma \cup \{\varepsilon\}))^*$ where $\mathbb{T} = (\mathbb{P} \cup \{a^r | a \in \mathbb{P}\} \cup \{a^l | a \in \mathbb{P}\})^*$
4. $S \in \mathbb{P}$ is the "start" type

Before giving more constraints, we need to define two operators on $\mathbb{T}$: merge and move; and functions on tuples.

$$\text{(merge)} \quad \begin{pmatrix} t_1 & \ldots & t_{k-1} \\ s_1 & \ldots & s_{k-1} \end{pmatrix} \bullet \begin{pmatrix} t_k & \ldots & t_n \\ s_k & \ldots & s_n \end{pmatrix} = \begin{pmatrix} t_1 & \ldots & t_n \\ s_1 & \ldots & s_n \end{pmatrix}$$

$$\text{(move)} \quad \begin{pmatrix} t_1 & \ldots & t_n \\ s_1 & \ldots & s_n \end{pmatrix}_{-i-j} = \begin{pmatrix} t_i t_j \\ s_i s_j \end{pmatrix} \bullet \begin{pmatrix} t_1 & \ldots & t_{i-1} & t_{i+1} & \ldots & t_n \\ s_1 & \ldots & s_{i-1} & s_{i+1} & \ldots & s_n \end{pmatrix}$$

Then we define two functions: GCON and GEXP (the latter will not be used in what follows)

$$\text{(GCON1)} \quad \begin{pmatrix} \cdots & xa^lby & \cdots \\ & s & \end{pmatrix} \to \begin{pmatrix} \cdots & xy & \cdots \\ & s & \end{pmatrix} \quad \text{only if } b \leq a$$

$$\text{(GCON2)} \quad \begin{pmatrix} \cdots & xab^ry & \cdots \\ & s & \end{pmatrix} \to \begin{pmatrix} \cdots & xy & \cdots \\ & s & \end{pmatrix} \quad \text{only if } a \leq b$$

$$\text{(GEXP1)} \quad \begin{pmatrix} \cdots & xy & \cdots \\ & s & \end{pmatrix} \to \begin{pmatrix} \cdots & xa^lby & \cdots \\ & s & \end{pmatrix} \quad \text{for } b \leq a$$

$$(\text{GEXP1}) \qquad \begin{pmatrix} \cdots & xy & \cdots \\ & s & \end{pmatrix} \to \begin{pmatrix} \cdots & xab^r y & \cdots \\ & s & \end{pmatrix} \qquad \text{for } a \leq b$$

As in [11], we add "performance restriction": We say that a tuple is *proper* iff:

- every type in it has exactly one atom
- no two types have the same atom
- GCON does not apply to it or to any result of applying Move to it.

and a tuple is saturated iff every type in it is saturated (i.e no adjoint types occur in it).

Then, the constraints are:

- (merge) applies to a pair of tuples only if both are proper and at least one is saturated
- (move) applies to a pair of typed strings in a tuple only if the types are both proper and at least one of them is saturated

The lexicon also has to be proper.

Finally, we get the following result (Theorem 3 in [11]):

Theorem 2.5 *: mTPG and MCFG are weakly-equivalent*

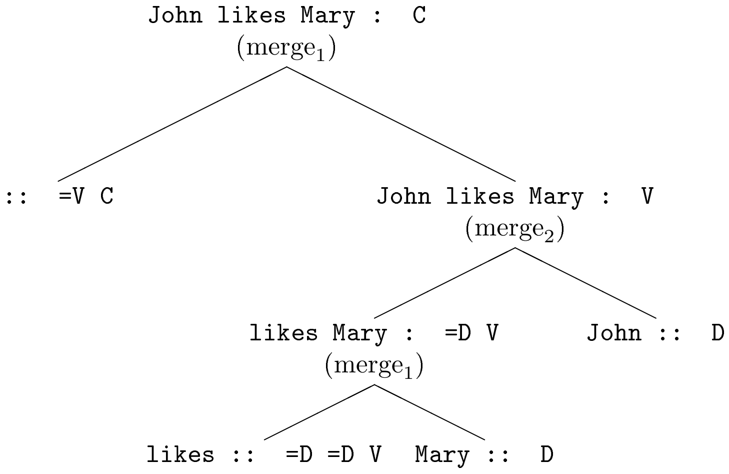
3 Examples and conversion into MCFG

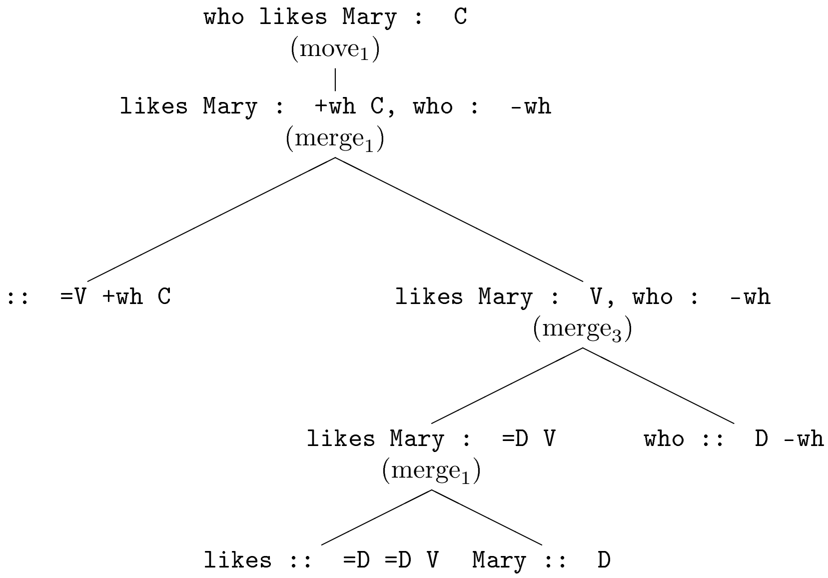
3.1 MG

To better understand the way the conversion will be operated, it is interesting to see how derivations occur with a minimalist grammar. Consider the following lexicon, with `C` being the start category:

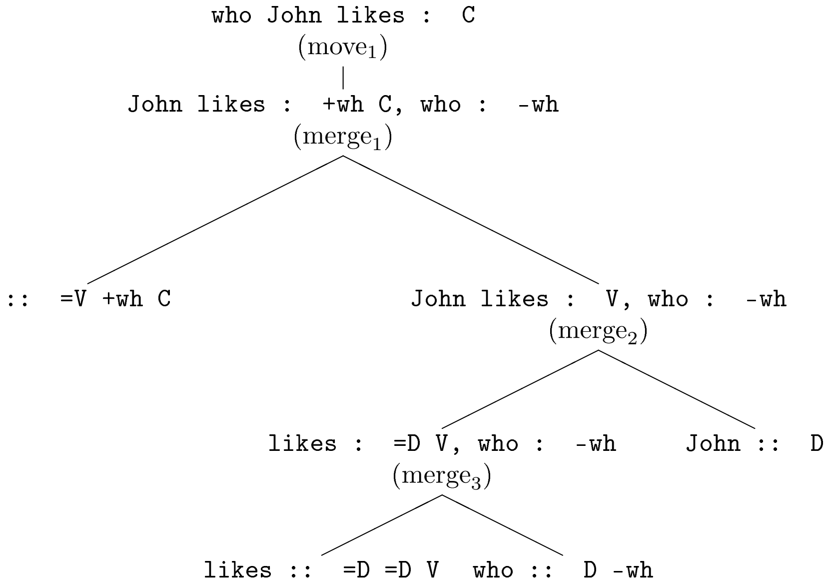
```
Mary :: D
John :: D
likes :: =D =D V
who :: D -wh
:: =V C
:: =V +wh C
```

Consider the sentences "John likes Mary", "who likes Mary" and "who John likes". They are all recognized by the grammar, with the following derivation trees:





And finally:



The remarkable facts are:

- The nodes have one or two branches: this is comforting considering the binary form of m-MCFG we want to produce.
- We do see tuples and mapping function appear, and we observe that the possibility of branching depends only on the involved expressions and the type of operation.

So we can easily imagine a rewriting system corresponding to every step of derivation. Let's assume we have a one-to-one function σ associating to every expression e a symbol $\sigma(e)$, then we can find MCFG rules for every function merge$_1$, merge$_2$, merge$_3$, move$_1$ and move$_2$:

merge1

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

corresponds to the following rule (with the convention that if $k = 0$, the arity of the map function is 1):

$$\sigma(\gamma, \alpha_1, \ldots, \alpha_k) \to \sigma(= f\gamma) \quad \sigma(f, \alpha_1, \ldots, \alpha_k) \quad [0,0;1,0][0,1]\ldots[0,k] \quad \text{(a binary 2-MCFG rule)}$$

Of course, we have to add rules to specify the string values in the lexicon:

$$s :: \delta$$

becomes

$$\sigma(\delta) \to s \qquad \text{(a terminating 2-MCFG rule)}$$

Similarly:

merge2

$$\frac{s := f\gamma, \alpha_1, \ldots, \alpha_k \qquad t \cdot f, \beta_1, \ldots, \beta_l}{ts : \gamma, \alpha_1, \ldots, \alpha_k, \beta_1, \ldots, \beta_l}$$

$$\Downarrow$$

$$\sigma(\gamma, \alpha_1, \ldots, \alpha_k, \beta_1, \ldots, \beta_l) \to \sigma(= f\gamma, \alpha_1, \ldots, \alpha_k) \quad \sigma(f, \beta_1, \ldots, \beta_l) \quad [1,0;0,0][0,1]\ldots[0,k][1,0]\ldots[1,l]$$

merge3

$$\frac{s\cdot = f\gamma, \alpha_1, \ldots, \alpha_k \qquad t \cdot f\delta, \beta_1, \ldots, \beta_l}{s : \gamma, \alpha_1, \ldots, \alpha_k, t : \delta, \beta_1, \ldots, \beta_l}$$

$$\Downarrow$$

$$\sigma(\gamma, \alpha_1, \ldots, \alpha_k, \delta, \beta_1, \ldots, \beta_l) \to \sigma(= f\gamma, \alpha_1, \ldots, \alpha_k) \quad \sigma(f\delta, \beta_1, \ldots, \beta_l) \quad [0,0]\ldots[0,k][1,0]\ldots[1,l]$$

move1

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

$$\Downarrow$$

$$\sigma(\gamma, \alpha_1, \ldots, \alpha_{i-1}, \alpha_{i+1}, \ldots, \alpha_k) \to \sigma(+f\gamma, \alpha_1, \ldots, \alpha_{i-1}, -f, \alpha_{i+1}, \ldots, \alpha_k)$$

$$[0,i;0,0][0,1]\ldots[0,i-1][0,i+1]\ldots[0,k]$$

move2

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

$$\Downarrow$$

$$\sigma(\gamma, \alpha_1, \ldots, \alpha_{i-1}, \delta, \alpha_{i+1}, \ldots, \alpha_k) \to \sigma(+f\gamma, \alpha_1, \ldots, \alpha_{i-1}, -f\delta, \alpha_{i+1}, \ldots, \alpha_k) \quad [0,0]\ldots[0,k]$$

Finally, for any start symbol C, we need to add the rule

$$S \to \sigma(C) \quad [0,0]$$

Overall, converting a MG lexicon to a MCFG is closing this lexicon with the merge and the move functions, producing at each successful inference step the corresponding MCFG rule, and finally adding the terminating rules as well as the starting rules.

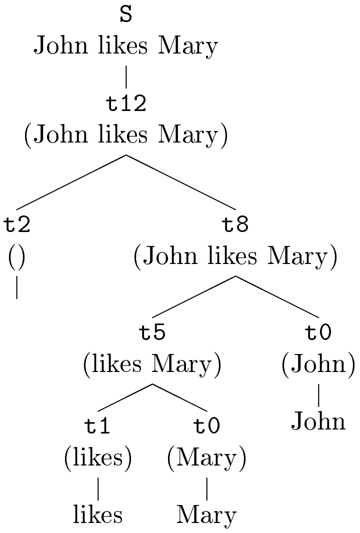
Applying this closure to our example gives us:

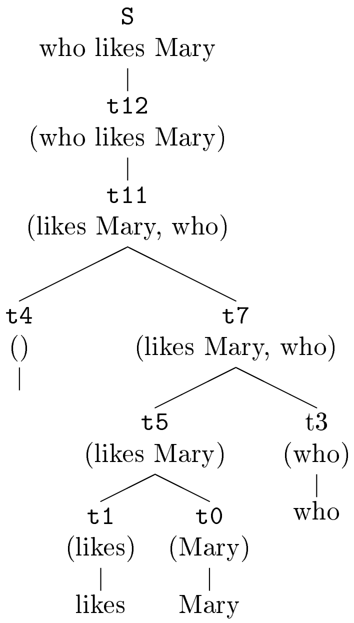
```
S --> t12 [0,0]
t12 --> t11 [0,1;0,0] (* move *)
t11 --> t4 t7 [0,0;1,0][1,1] (* merge1 *)
t4 --> ""
t7 --> t5 t3 [0,0][1,0] (* merge3 *)
t5 --> t1 t0 [0,0;1,0] (* merge1 *)
t1 --> "likes"
t0 --> "Mary"
t0 --> "John"
t3 --> "who"
t7 --> t6 t0 [1,0;0,0][0,1] (* merge2 *)
t6 --> t1 t3 [0,0][1,0] (* merge3 *)
t12 --> t2 t8 [0,0;1,0] (* merge1 *)
t2 --> ""
t8 --> t5 t0 [1,0;0,0] (* merge2 *)
```

Where the symbols correspond to:

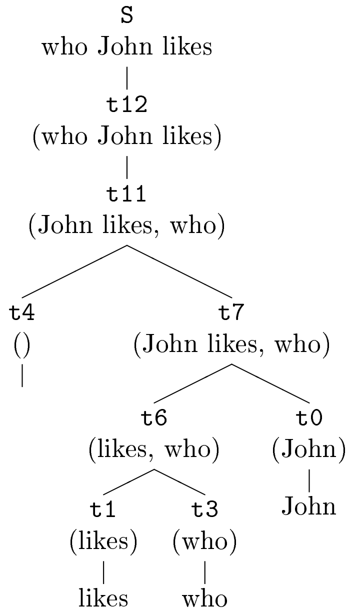
```
t0 : (:: D)
t1 : (:: =D =D V)
t2 : (:: =V C)
t3 : (:: D -wh)
t4 : (:: =V +wh C)
t5 : (: =D V)
t6 : (: =D V;: -wh)
t7 : (: V;: -wh)
t8 : (: V)
t11 : (: +wh C;: -wh)
t12 : (: C)
```

If we try to derive the same sentences as above, we obtain:





And finally:



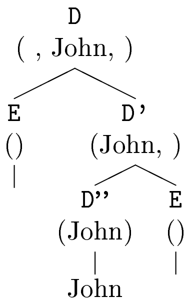
Except for the leaves and the root, we follow exactly the derivations of the MG system.

3.2 hMG

Dealing with hMG is similar. To satisfy the normal form requirements of our target MCFGs, we need to be careful though with lexical items: for a rule like `John :: D` a scheme like

```
E --> ""
D'' --> "John"
D' --> D'' E [0,0][1,0]
D --> E D' [0,0][1,0][1,1]
```

creates the good symbol `D` of arity 3, containing only a non-empty head, but empty first and third components. In the general case, and instead of `D`, we use $\sigma(D)$.



Other change, for starting rules, we have to concatenate the 3-tuple: (example where C is a start category)

$$S \rightarrow \sigma(C) \quad [0,0;0,1;0,2]$$

We can now give the MCFG rules corresponding to each of the 17 partial functions of the hMG framework.

r1'

$$\frac{\varepsilon, s, \varepsilon ::= f\gamma \qquad t_s, t_h, t_c \cdot f, \alpha_1, \ldots, \alpha_k}{\varepsilon, s, t_s t_h t_c : \gamma, \alpha_1, \ldots, \alpha_k}$$

$$\Downarrow$$

$$\sigma(\gamma, \alpha_1, \ldots, \alpha_k) \rightarrow \sigma(= f\gamma) \quad \sigma(f, \alpha_1, \ldots, \alpha_k) \quad [0,0][0,1][0,2;1,0;1,1;1,2][1,3]\ldots[1,k+2]$$

r1right

$$\frac{\varepsilon, s, \varepsilon ::<= f\gamma \qquad t_s, t_h, t_c \cdot f, \alpha_1, \ldots, \alpha_k}{\varepsilon, s t_h, t_s t_c : \gamma, \alpha_1, \ldots, \alpha_k}$$

$$\Downarrow$$

$$\sigma(\gamma, \alpha_1, \ldots, \alpha_k) \rightarrow \sigma(<= f\gamma) \quad \sigma(f, \alpha_1, \ldots, \alpha_k) \quad [0,0][0,1;1,1][0,2;1,0;1,2][1,3]\ldots[1,k+2]$$

r1left

$$\frac{\varepsilon, s, \varepsilon ::=> f\gamma \qquad t_s, t_h, t_c \cdot f, \alpha_1, \ldots, \alpha_k}{\varepsilon, t_h s, t_s t_c : \gamma, \alpha_1, \ldots, \alpha_k}$$

$$\Downarrow$$

$$\sigma(\gamma, \alpha_1, \ldots, \alpha_k) \rightarrow \sigma(=> f\gamma) \quad \sigma(f, \alpha_1, \ldots, \alpha_k) \quad [0,0][1,1;0,1][0,2;1,0;1,2][1,3]\ldots[1,k+2]$$

r2'

$$\frac{s_s, s_h, s_c := f\gamma, \alpha_1, \ldots, \alpha_k \qquad t_s, t_h, t_c \cdot f, \beta_1, \ldots, \beta_l}{t_s t_h t_c s_s, s_h, s_c : \gamma, \alpha_1, \ldots, \alpha_k, \beta_1, \ldots, \beta_l}$$

$$\Downarrow$$

$$\sigma(\gamma, \alpha_1, \ldots, \alpha_k, \beta_1, \ldots, \beta_l) \rightarrow \sigma(= f\gamma, \alpha_1, \ldots, \alpha_k) \quad \sigma(f, \beta_1, \ldots, \beta_l)$$

$$[1,0;1,1;1,2;0,0][0,1][0,2][0,3]\ldots[0,k+2][1,3]\ldots[1,l+2]$$

r3'

$$\frac{s_s, s_h, s_c \cdot = f\gamma, \alpha_1, \ldots, \alpha_k \qquad t_s, t_h, t_c \cdot f\delta, \beta_1, \ldots, \beta_l}{s_s, s_h, s_c : \gamma, \alpha_1, \ldots, \alpha_k, t_s t_h t_c : \delta, \beta_1, \ldots, \beta_l}$$

$$\Downarrow$$

$$\sigma(\gamma, \alpha_1, \ldots, \alpha_k, \delta, \beta_1, \ldots, \beta_l) \rightarrow \sigma(= f\gamma, \alpha_1, \ldots, \alpha_k) \quad \sigma(f\delta, \beta_1, \ldots, \beta_l)$$

$$[0,0]\ldots[0,k+2][1,0;1,1;1,2][1,3]\ldots[1,l+2]$$

r3right

$$\frac{s_s, s_h, s_c \cdot <= f\gamma, \alpha_1, \ldots, \alpha_k \qquad t_s, t_h, t_c \cdot f\delta, \beta_1, \ldots, \beta_l}{s_s, s_h t_h, s_c : \gamma, \alpha_1, \ldots, \alpha_k, t_s t_c : \delta, \beta_1, \ldots, \beta_l}$$

$$\Downarrow$$

$$\sigma(\gamma, \alpha_1, \ldots, \alpha_k, \delta, \beta_1, \ldots, \beta_l) \rightarrow \sigma(<= f\gamma, \alpha_1, \ldots, \alpha_k) \quad \sigma(f\delta, \beta_1, \ldots, \beta_l)$$

$$[0,0][0,1;1,1][0,2]\ldots[0,k+2][1,0;1,2][1,3]\ldots[1,l+2]$$

r3left

$$\frac{s_s, s_h, s_c \cdot <= f\gamma, \alpha_1, \ldots, \alpha_k \qquad t_s, t_h, t_c \cdot f\delta, \beta_1, \ldots, \beta_l}{s_s, t_h s_h, s_c : \gamma, \alpha_1, \ldots, \alpha_k, t_s t_c : \delta, \beta_1, \ldots, \beta_l}$$

$$\Downarrow$$

$$\sigma(\gamma, \alpha_1, \ldots, \alpha_k, \delta, \beta_1, \ldots, \beta_l) \rightarrow \sigma(<= f\gamma, \alpha_1, \ldots, \alpha_k) \quad \sigma(f\delta, \beta_1, \ldots, \beta_l)$$

$$[0,0][1,1;0,1][0,2]\ldots[0,k+2][1,0;1,2][1,3]\ldots[1,l+2]$$

r1hopright

$$\frac{\varepsilon, s, \varepsilon ::==> f\gamma \qquad t_s, t_h, t_c \cdot f, \alpha_1, \ldots, \alpha_k}{\varepsilon, \varepsilon, t_s t_h s t_c : \gamma, \alpha_1, \ldots, \alpha_k}$$

$$\Downarrow$$

$$\sigma(\gamma, \alpha_1, \ldots, \alpha_k) \rightarrow \sigma(==> f\gamma) \quad \sigma(f, \alpha_1, \ldots, \alpha_k)$$

$$[0,0][0,2][1,0;1,1;0,1;1,2][1,3]\ldots[1,k+2]$$

r1hopleft

$$\frac{\varepsilon, s, \varepsilon ::<== f\gamma \qquad t_s, t_h, t_c \cdot f, \alpha_1, \ldots, \alpha_k}{\varepsilon, \varepsilon, t_s s t_h t_c : \gamma, \alpha_1, \ldots, \alpha_k}$$

$$\Downarrow$$

$$\sigma(\gamma, \alpha_1, \ldots, \alpha_k) \rightarrow \sigma(<== f\gamma) \quad \sigma(f, \alpha_1, \ldots, \alpha_k)$$

$$[0,0][0,2][1,0;0,1;1,1;1,2][1,3]\ldots[1,k+2]$$

r3hopright

$$\frac{\varepsilon, s, \varepsilon ::==> f\gamma, \alpha_1, \ldots, \alpha_k \qquad t_s, t_h, t_c \cdot f\delta, \beta_1, \ldots, \beta_l}{\varepsilon, \varepsilon, \varepsilon : \gamma, \alpha_1, \ldots, \alpha_k, t_s t_h s t_c : \delta, \beta_1, \ldots, \beta_l}$$

$$\Downarrow$$

$$\sigma'(\gamma, \alpha_1, \ldots, \alpha_k, \delta, \beta_1, \ldots, \beta_l) \rightarrow \sigma(==> f\gamma, \alpha_1, \ldots, \alpha_k) \quad \sigma(f\delta, \beta_1, \ldots, \beta_l)$$

$$[0,0][0,2]\ldots[0,k+2][1,0;1,1;0,1;1,2][1,3]\ldots[1,l+2]$$

as well as

$$\sigma(\gamma, \alpha_1, \ldots, \alpha_k, \delta, \beta_1, \ldots, \beta_l) \rightarrow E \quad \sigma'(\gamma, \alpha_1, \ldots, \alpha_k, \delta, \beta_1, \ldots, \beta_l) \quad [0,0][1,0]\ldots[1,k+l+2]$$

where σ' is another coding function whose range is disjoint from the range of σ

r3hopleft

$$\frac{\varepsilon, s, \varepsilon ::<== f\gamma, \alpha_1, \ldots, \alpha_k \qquad t_s, t_h, t_c \cdot f\delta, \beta_1, \ldots, \beta_l}{\varepsilon, \varepsilon, \varepsilon : \gamma, \alpha_1, \ldots, \alpha_k, t_s s t_h t_c : \delta, \beta_1, \ldots, \beta_l}$$

$$\Downarrow$$

$$\begin{array}{rcl} \sigma''(\gamma, \alpha_1, \ldots, \alpha_k, \delta, \beta_1, \ldots, \beta_l) & \rightarrow & \sigma(<== f\gamma, \alpha_1, \ldots, \alpha_k) \quad \sigma(f\delta, \beta_1, \ldots, \beta_l) \\ & & [0,0][0,2]\ldots[0,k+2][1,0;0,1;1,1;1,2][1,3]\ldots[1,l+2] \\ \sigma(\gamma, \alpha_1, \ldots, \alpha_k, \delta, \beta_1, \ldots, \beta_l) & \rightarrow & E \quad \sigma'(\gamma, \alpha_1, \ldots, \alpha_k, \delta, \beta_1, \ldots, \beta_l) \quad [0,0][1,0]\ldots[1,k+l+2] \end{array}$$

where σ'' is a coding function whose range is disjoint from the ranges of σ and σ'

m1’

$$\frac{s_s, s_h, s_c : +f\gamma, \alpha 1, \ldots, \alpha_{i-1}, t : -f, \alpha_{i+1}, \ldots, \alpha_k}{ts_s, s_h, s_c : \gamma, \alpha 1, \ldots, \alpha_{i-1}, \alpha_{i+1}, \ldots, \alpha_k}$$

$$\Downarrow$$

$$\sigma(\gamma, \alpha 1, \ldots, \alpha_{i-1}, \alpha_{i+1}, \ldots, \alpha_k) \to \sigma(+f\gamma, \alpha 1, \ldots, \alpha_{i-1}, -f, \alpha_{i+1}, \ldots, \alpha_k)$$

$$[0, i; 0, 0][0, 1] \ldots [0, i-1][0, i+1] \ldots [0, k+2]$$

m2’

$$\frac{s_s, s_h, s_c : +f\gamma, \alpha 1, \ldots, \alpha_{i-1}, t : -f\delta, \alpha_{i+1}, \ldots, \alpha_k}{s_s, s_h, s_c : \gamma, \alpha 1, \ldots, \alpha_{i-1}, t : \delta, \alpha_{i+1}, \ldots, \alpha_k}$$

$$\Downarrow$$

$$\sigma(\gamma, \alpha 1, \ldots, \alpha_{i-1}, \delta, \alpha_{i+1}, \ldots, \alpha_k) \to \sigma(+f\gamma, \alpha 1, \ldots, \alpha_{i-1}, -f\delta, \alpha_{i+1}, \ldots, \alpha_k) \quad [0, 0] \ldots [0, k+2]$$

left-adjoin1 **if** $f\gamma \gg g\eta$

$$\frac{s_s, s_h, s_c \cdot f\gamma, \alpha_1, \ldots, \alpha_k \qquad t_s, t_h, t_c \cdot g\eta\nu, \beta_1, \ldots, \beta_l}{s_s s_h s_c t_s, t_h, t_c : g\eta\nu, \alpha_1, \ldots, \alpha_k, \beta_1, \ldots, \beta_l}$$

$$\Downarrow$$

$$\sigma(g\eta\nu, \alpha_1, \ldots, \alpha_k, \beta_1, \ldots, \beta_l) \to \sigma(f\gamma, \alpha_1, \ldots, \alpha_k) \quad \sigma(g\eta\nu, \beta_1, \ldots, \beta_l)$$

$$[0, 0; 0, 1; 0, 2; 1, 0][1, 1][1, 2][0, 3] \ldots [0, k+2][1, 3] \ldots [1, l+2]$$

right-adjoin1 **if** $g\eta \ll f\gamma$

$$\frac{s_s, s_h, s_c \cdot f\gamma, \alpha_1, \ldots, \alpha_k \qquad t_s, t_h, t_c \cdot g\eta\nu, \beta_1, \ldots, \beta_l}{t_s, t_h, t_c s_s s_h s_c : g\eta\nu, \alpha_1, \ldots, \alpha_k, \beta_1, \ldots, \beta_l}$$

$$\Downarrow$$

$$\sigma(g\eta\nu, \alpha_1, \ldots, \alpha_k, \beta_1, \ldots, \beta_l) \to \sigma(f\gamma, \alpha_1, \ldots, \alpha_k) \quad \sigma(g\eta\nu, \beta_1, \ldots, \beta_l)$$

$$[1, 0][1, 1][1, 2; 0, 0; 0, 1; 0, 2][0, 3] \ldots [0, k+2][1, 3] \ldots [1, l+2]$$

left-adjoin2 **if** $f\gamma \gg g\eta$

$$\frac{s_s, s_h, s_c \cdot f\gamma\delta, \alpha_1, \ldots, \alpha_k \qquad t_s, t_h, t_c \cdot g\eta\nu, \beta_1, \ldots, \beta_l}{t_s, t_h, t_c : g\eta\nu, s_s s_h s_c : \delta, \alpha_1, \ldots, \alpha_k, \beta_1, \ldots, \beta_l}$$

$$\Downarrow$$

$$\sigma(g\eta\nu, \delta, \alpha_1, \ldots, \alpha_k, \beta_1, \ldots, \beta_l) \to \sigma(f\gamma\delta, \alpha_1, \ldots, \alpha_k) \quad \sigma(g\eta\nu, \beta_1, \ldots, \beta_l)$$

$$[1, 0][1, 1][1, 2][0, 0; 1, 0; 2, 0][0, 3] \ldots [0, k+2][1, 3] \ldots [1, l+2]$$

right-adjoin2 **if** $g\eta \ll f\gamma$

$$\frac{s_s, s_h, s_c \cdot f\gamma\delta, \alpha_1, \dots, \alpha_k \qquad t_s, t_h, t_c \cdot g\eta\nu, \beta_1, \dots, \beta_l}{t_s, t_h, t_c : g\eta\nu, s_s s_h s_c : \delta, \alpha_1, \dots, \alpha_k, \beta_1, \dots, \beta_l}$$

$$\Downarrow$$

$$\sigma(g\eta\nu, \delta, \alpha_1, \dots, \alpha_k, \beta_1, \dots, \beta_l) \rightarrow \sigma(f\gamma\delta, \alpha_1, \dots, \alpha_k) \quad \sigma(g\eta\nu, \beta_1, \dots, \beta_l)$$

$$[1,0][1,1][1,2][0,0;1,0;2,0][0,3]\dots[0,k+2][1,3]\dots[1,l+2]$$

To convert a hMG to a MCFG, we proceed the same way than for MG: we close the hMG with all the functions and at each step we obtain a MCFG rule which is to add to the destination set of MCFG rules.

Consider the following example for french clitics:

```
/start symbol/
C;
/rules/
:: =T C;
:: =Refl12 +k T;
:: =Acc3 +k T;
:: =Dat3 +k T;
:: =v +k T;
se :: =Acc3 +F Refl12;
se :: =Dat3 +F Refl12;
se :: =v +F Refl12;
le :: =Dat3 +G Acc3;
le :: =v +G Acc3;
lui :: =v +F Dat3;
:: <=vacc =D v;
:: <=vdat =D +k vacc;
:: <=V =p vdat;
montrera :: V;
:: <=P p;
a :: =D +k P;
:: p -F;
Jean :: D -k;
Marie :: D -k;
le :: =N D -k;
:: D -k -F;
:: D -k -G;
roi :: N;
livre :: N;
```

The closure of this grammar gives:

```
S --> t172 [0,0;0,1;0,2] (* concatenation *)
t172 --> t0 t168 [0,0][0,1][0,2;1,0;1,1;1,2] (* r1' *)
t0 --> E t0_tmp2 [0,0][1,0][1,1]
t0_tmp2 --> t0_tmp1 E [0,0][1,0]
t0_tmp1 --> ""
t168 --> t129 [0,3;0,0][0,1][0,2] (* move' *)
t129 --> t2 t169 [0,0][0,1][0,2;1,0;1,1;1,2][1,3] (* r1' *)
t2 --> E t2_tmp2 [0,0][1,0][1,1]
t2_tmp2 --> t2_tmp1 E [0,0][1,0]
t2_tmp1 --> ""
```

```
t169 --> t138 [0,3;0,0][0,1][0,2][0,4] (* move' *)
t138 --> t9 t96 [0,0][0,1][0,2;1,0;1,1;1,2][1,3][1,4] (* r1' *)
t9 --> E t9_tmp2 [0,0][1,0][1,1]
t9_tmp2 --> t9_tmp1 E [0,0][1,0]
t9_tmp1 --> "le"
t96 --> t90 t18 [0,0][0,1][0,2][0,3][1,0;1,1;1,2] (* r3' *)
t90 --> t11 t63 [0,0][0,1;1,1][0,2;1,0;1,2][1,3] (* r1right *)
t11 --> E t11_tmp2 [0,0][1,0][1,1]
t11_tmp2 --> t11_tmp1 E [0,0][1,0]
t11_tmp1 --> ""
t63 --> t47 [0,0][0,1][0,2][0,3] (* move' *)
t47 --> t45 t21 [0,0][0,1][0,2][1,0;1,1;1,2] (* r3' *)
t45 --> t12 t39 [0,0][0,1;1,1][0,2;1,0;1,2] (* r1right *)
t12 --> E t12_tmp2 [0,0][1,0][1,1]
t12_tmp2 --> t12_tmp1 E [0,0][1,0]
t12_tmp1 --> ""
t39 --> t27 t35 [1,0;1,1;1,2;0,0][0,1][0,2] (* r2' *)
t27 --> t13 t14 [0,0][0,1;1,1][0,2;1,0;1,2] (* r1right *)
t13 --> E t13_tmp2 [0,0][1,0][1,1]
t13_tmp2 --> t13_tmp1 E [0,0][1,0]
t13_tmp1 --> ""
t14 --> E t14_tmp2 [0,0][1,0][1,1]
t14_tmp2 --> t14_tmp1 E [0,0][1,0]
t14_tmp1 --> "montrera"
t35 --> t15 t29 [0,0][0,1;1,1][0,2;1,0;1,2] (* r1right *)
t15 --> E t15_tmp2 [0,0][1,0][1,1]
t15_tmp2 --> t15_tmp1 E [0,0][1,0]
t15_tmp1 --> ""
t29 --> t25 [0,3;0,0][0,1][0,2] (* move' *)
t25 --> t16 t26 [0,0][0,1][0,2][1,0;1,1;1,2] (* r3' *)
t16 --> E t16_tmp2 [0,0][1,0][1,1]
t16_tmp2 --> t16_tmp1 E [0,0][1,0]
t16_tmp1 --> "a"
t26 --> t19 t22 [0,0][0,1][0,2;1,0;1,1;1,2] (* r1' *)
t19 --> E t19_tmp2 [0,0][1,0][1,1]
t19_tmp2 --> t19_tmp1 E [0,0][1,0]
t19_tmp1 --> "le"
t22 --> E t22_tmp2 [0,0][1,0][1,1]
t22_tmp2 --> t22_tmp1 E [0,0][1,0]
t22_tmp1 --> "livre"
t22_tmp1 --> "roi"
t25 --> t16 t18 [0,0][0,1][0,2][1,0;1,1;1,2] (* r3' *)
t18 --> E t18_tmp2 [0,0][1,0][1,1]
t18_tmp2 --> t18_tmp1 E [0,0][1,0]
t18_tmp1 --> "Marie"
t18_tmp1 --> "Jean"
t21 --> E t21_tmp2 [0,0][1,0][1,1]
t21_tmp2 --> t21_tmp1 E [0,0][1,0]
t21_tmp1 --> ""
t63 --> t49 [0,4;0,0][0,1][0,2][0,3] (* move' *)
t49 --> t44 t18 [0,0][0,1][0,2][0,3][1,0;1,1;1,2] (* r3' *)
t44 --> t12 t40 [0,0][0,1;1,1][0,2;1,0;1,2][1,3] (* r1right *)
t40 --> t27 t33 [1,0;1,1;1,2;0,0][0,1][0,2][1,3] (* r2' *)
t33 --> t15 t31 [0,0][0,1;1,1][0,2;1,0;1,2][1,3] (* r1right *)
```

```
t31 --> t23 [0,0][0,1][0,2][0,3] (* move' *)
t23 --> t16 t21 [0,0][0,1][0,2][1,0;1,1;1,2] (* r3' *)
t49 --> t44 t26 [0,0][0,1][0,2][0,3][1,0;1,1;1,2] (* r3' *)
t96 --> t90 t26 [0,0][0,1][0,2][0,3][1,0;1,1;1,2] (* r3' *)
t138 --> t8 t113 [0,0][0,1][0,2;1,0;1,1;1,2][1,3][1,4] (* r1' *)
t8 --> E t8_tmp2 [0,0][1,0][1,1]
t8_tmp2 --> t8_tmp1 E [0,0][1,0]
t8_tmp1 --> "le"
t113 --> t153 [0,4;0,0][0,1][0,2][0,3][0,5] (* move' *)
t153 --> t10 t93 [0,0][0,1][0,2;1,0;1,1;1,2][1,3][1,4][1,5] (* r1' *)
t10 --> E t10_tmp2 [0,0][1,0][1,1]
t10_tmp2 --> t10_tmp1 E [0,0][1,0]
t10_tmp1 --> "lui"
t93 --> t92 t18 [0,0][0,1][0,2][0,3][0,4][1,0;1,1;1,2] (* r3' *)
t92 --> t11 t61 [0,0][0,1;1,1][0,2;1,0;1,2][1,3][1,4] (* r1right *)
t61 --> t51 [0,0][0,1][0,2][0,3][0,4] (* move' *)
t51 --> t44 t20 [0,0][0,1][0,2][0,3][1,0;1,1;1,2] (* r3' *)
t20 --> E t20_tmp2 [0,0][1,0][1,1]
t20_tmp2 --> t20_tmp1 E [0,0][1,0]
t20_tmp1 --> ""
t93 --> t92 t26 [0,0][0,1][0,2][0,3][0,4][1,0;1,1;1,2] (* r3' *)
t113 --> t70 [0,3;0,0][0,1][0,2][0,4][0,5] (* move' *)
t70 --> t10 t55 [0,0][0,1][0,2;1,0;1,1;1,2][1,3][1,4][1,5] (* r1' *)
t55 --> t53 t18 [0,0][0,1][0,2][0,3][0,4][1,0;1,1;1,2] (* r3' *)
t53 --> t11 t42 [0,0][0,1;1,1][0,2;1,0;1,2][1,3][1,4] (* r1right *)
t42 --> t37 [0,0][0,1][0,2][0,3][0,4] (* move' *)
t37 --> t32 t21 [0,0][0,1][0,2][0,3][1,0;1,1;1,2] (* r3' *)
t32 --> t12 t28 [0,0][0,1;1,1][0,2;1,0;1,2][1,3] (* r1right *)
t28 --> t27 t34 [1,0;1,1;1,2;0,0][0,1][0,2][1,3] (* r2' *)
t34 --> t15 t30 [0,0][0,1;1,1][0,2;1,0;1,2][1,3] (* r1right *)
t30 --> t24 [0,0][0,1][0,2][0,3] (* move' *)
t24 --> t16 t20 [0,0][0,1][0,2][1,0;1,1;1,2] (* r3' *)
t28 --> t27 t17 [0,0][0,1][0,2][1,0;1,1;1,2] (* r3' *)
t17 --> E t17_tmp2 [0,0][1,0][1,1]
t17_tmp2 --> t17_tmp1 E [0,0][1,0]
t17_tmp1 --> ""
t55 --> t53 t26 [0,0][0,1][0,2][0,3][0,4][1,0;1,1;1,2] (* r3' *)
t129 --> t4 t99 [0,0][0,1][0,2;1,0;1,1;1,2][1,3] (* r1' *)
t4 --> E t4_tmp2 [0,0][1,0][1,1]
t4_tmp2 --> t4_tmp1 E [0,0][1,0]
t4_tmp1 --> ""
t99 --> t89 t18 [0,0][0,1][0,2][1,0;1,1;1,2] (* r3' *)
t89 --> t11 t64 [0,0][0,1;1,1][0,2;1,0;1,2] (* r1right *)
t64 --> t46 [0,3;0,0][0,1][0,2] (* move' *)
t46 --> t45 t18 [0,0][0,1][0,2][1,0;1,1;1,2] (* r3' *)
t46 --> t45 t26 [0,0][0,1][0,2][1,0;1,1;1,2] (* r3' *)
t99 --> t89 t26 [0,0][0,1][0,2][1,0;1,1;1,2] (* r3' *)
t129 --> t3 t116 [0,0][0,1][0,2;1,0;1,1;1,2][1,3] (* r1' *)
t3 --> E t3_tmp2 [0,0][1,0][1,1]
t3_tmp2 --> t3_tmp1 E [0,0][1,0]
t3_tmp1 --> ""
t116 --> t67 [0,3;0,0][0,1][0,2][0,4] (* move' *)
t67 --> t10 t58 [0,0][0,1][0,2;1,0;1,1;1,2][1,3][1,4] (* r1' *)
t58 --> t52 t18 [0,0][0,1][0,2][0,3][1,0;1,1;1,2] (* r3' *)
```

```
t52 --> t11 t43 [0,0][0,1;1,1][0,2;1,0;1,2][1,3] (* r1right *)
t43 --> t48 [0,0][0,1][0,2][0,3] (* move' *)
t48 --> t45 t20 [0,0][0,1][0,2][1,0;1,1;1,2] (* r3' *)
t43 --> t36 [0,4;0,0][0,1][0,2][0,3] (* move' *)
t36 --> t32 t18 [0,0][0,1][0,2][0,3][1,0;1,1;1,2] (* r3' *)
t36 --> t32 t26 [0,0][0,1][0,2][0,3][1,0;1,1;1,2] (* r3' *)
t58 --> t52 t26 [0,0][0,1][0,2][0,3][1,0;1,1;1,2] (* r3' *)
t129 --> t1 t110 [0,0][0,1][0,2;1,0;1,1;1,2][1,3] (* r1' *)
t1 --> E t1_tmp2 [0,0][1,0][1,1]
t1_tmp2 --> t1_tmp1 E [0,0][1,0]
t1_tmp1 --> ""
t110 --> t79 [0,3;0,0][0,1][0,2][0,4] (* move' *)
t79 --> t5 t119 [0,0][0,1][0,2;1,0;1,1;1,2][1,3][1,4] (* r1' *)
t5 --> E t5_tmp2 [0,0][1,0][1,1]
t5_tmp2 --> t5_tmp1 E [0,0][1,0]
t5_tmp1 --> "se"
t119 --> t156 [0,3;0,0][0,1][0,2][0,4][0,5] (* move' *)
t156 --> t9 t93 [0,0][0,1][0,2;1,0;1,1;1,2][1,3][1,4][1,5] (* r1' *)
t119 --> t76 [0,4;0,0][0,1][0,2][0,3][0,5] (* move' *)
t76 --> t9 t55 [0,0][0,1][0,2;1,0;1,1;1,2][1,3][1,4][1,5] (* r1' *)
t79 --> t7 t58 [0,0][0,1][0,2;1,0;1,1;1,2][1,3][1,4] (* r1' *)
t7 --> E t7_tmp2 [0,0][1,0][1,1]
t7_tmp2 --> t7_tmp1 E [0,0][1,0]
t7_tmp1 --> "se"
E --> ""
```

3.3 mTPG

Considering the constraints discussed in section 2.4, mTPG can be considered as having only two functions: Conc and Gcon, having the following behavior:

$$\text{(Conc)} \qquad \begin{pmatrix} t_1 & \dots & t_{k-1} \\ s_1 & \dots & s_{k-1} \end{pmatrix} \bullet \begin{pmatrix} t_k & \dots & t_n \\ s_k & \dots & s_n \end{pmatrix} = \begin{pmatrix} t_1 & \dots & t_n \\ s_1 & \dots & s_n \end{pmatrix}$$

$$\text{(Gcon)}_{i,j}^{\leftarrow} \qquad \begin{pmatrix} t_1 & \dots & t_i & \dots & t_j & \dots & t_n \\ s_1 & \dots & s_i & \dots & s_j & \dots & s_n \end{pmatrix} \to \begin{pmatrix} t_1 & \dots & t_i t_j & \dots & t_n \\ s_1 & \dots & s_i s_j & \dots & s_n \end{pmatrix}$$

$$\text{(Gcon)}_{i,j}^{\rightarrow} \qquad \begin{pmatrix} t_1 & \dots & t_i & \dots & t_j & \dots & t_n \\ s_1 & \dots & s_i & \dots & s_j & \dots & s_n \end{pmatrix} \to \begin{pmatrix} t_1 & \dots & t_j t_i & \dots & t_n \\ s_1 & \dots & s_j s_i & \dots & s_n \end{pmatrix}$$

Where $i, j \in [1, n]$, and:

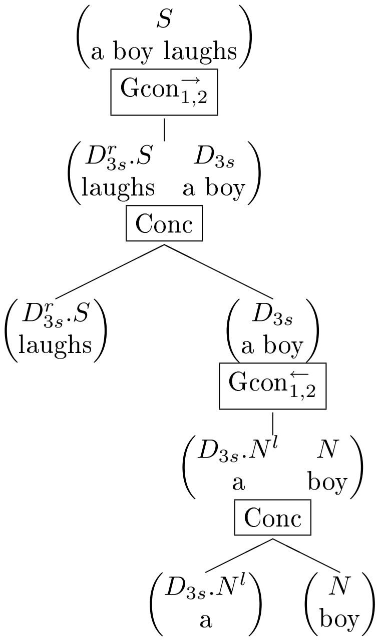
- In the case "$\leftarrow$", t_j is saturated and $t_i = t'_i \delta^l$ with $t_j \leq \delta$
- In the case "$\rightarrow$", t_j is saturated and $t_i = \delta^r t'_i$ with $t_j \leq \delta$

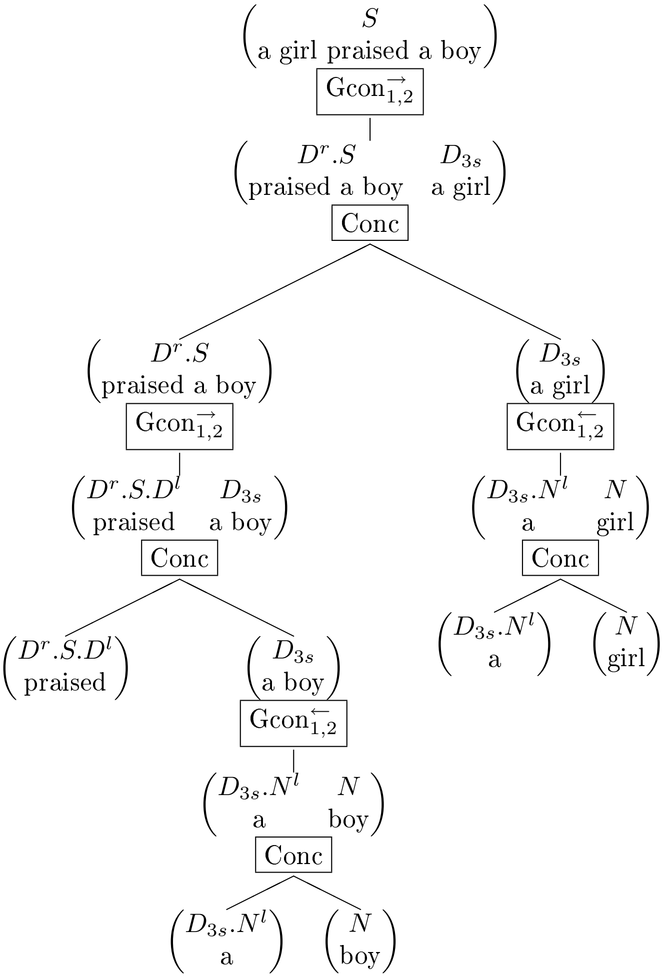
We can therefore operate GCON1 and GCON2 immediately, and that operation becomes transparent ($\leftarrow$: $t_i t_j = t'_i.\delta^l.t_j = t'_i$ and $\rightarrow$: $t_j t_i = t_j.\delta^r.t'_i = t'_i$).

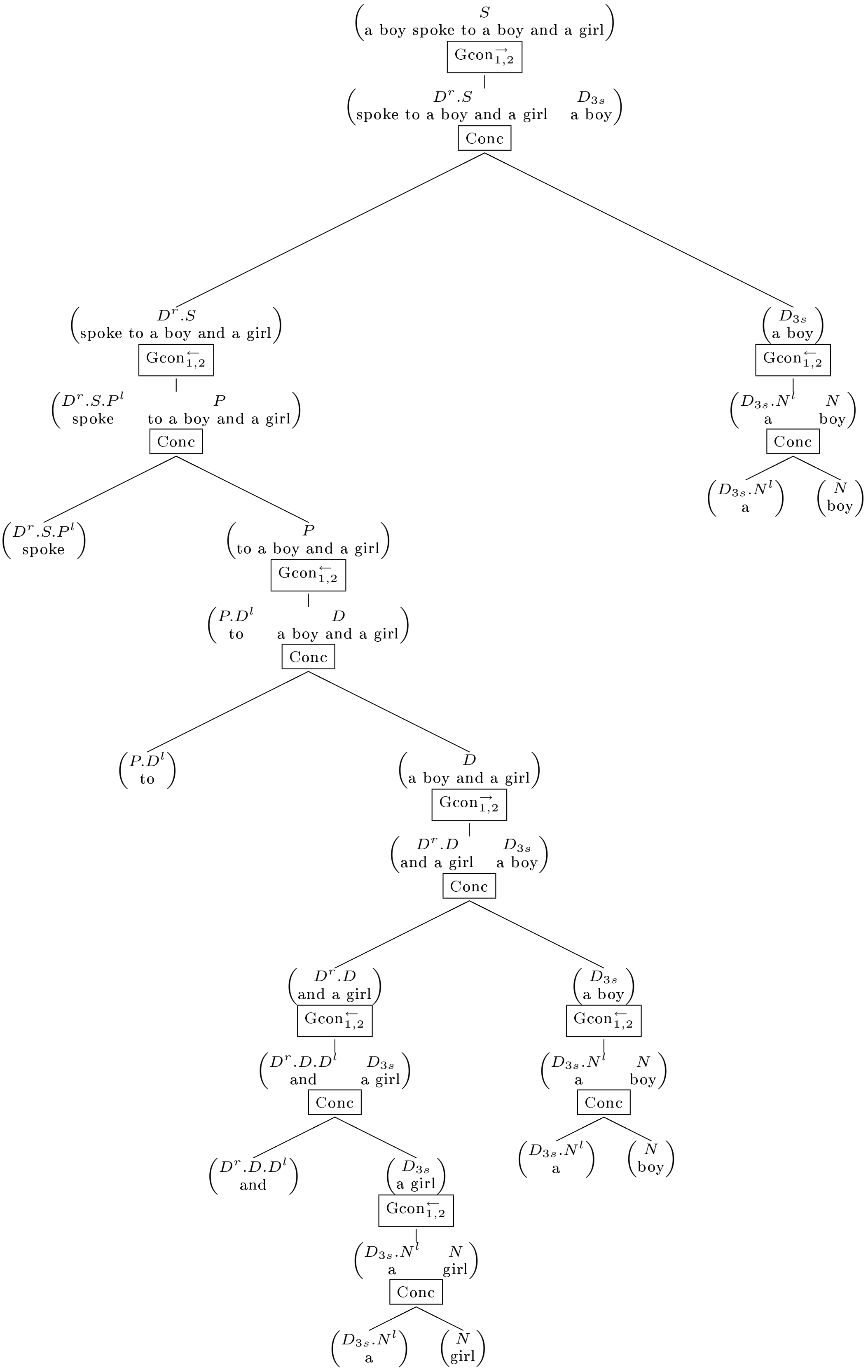
Let's see how derivations are done in this system. Consider the following grammar (from [11]), with S the starting type, and with $\leq$ being equality union $D_{3s} \leq D$:

$$\begin{pmatrix} D_{3s}.N^l \\ \text{a} \end{pmatrix} \begin{pmatrix} N \\ \text{boy} \end{pmatrix} \begin{pmatrix} N \\ \text{girl} \end{pmatrix} \begin{pmatrix} D^r.D.D^l \\ \text{and} \end{pmatrix} \begin{pmatrix} D_{3s}^r.S \\ \text{laughs} \end{pmatrix} \begin{pmatrix} D^r.S.D^l \\ \text{praised} \end{pmatrix} \begin{pmatrix} D^r.S.P^l \\ \text{spoke} \end{pmatrix} \begin{pmatrix} P.D^l \\ \text{to} \end{pmatrix}$$

Sentences like "a boy laughs", "a girl praised a boy" or "a boy spoke to a boy and a girl" are recognized with the following trees:


S a boy laughs
$\text{Gcon}^{\rightarrow}_{1,2}$
$D^r_{3s}.S$ laughs D_{3s} a boy
Conc
$D^r_{3s}.S$ laughs
D_{3s} a boy
$\text{Gcon}^{\leftarrow}_{1,2}$
$D_{3s}.N^l$ a N boy
Conc
$D_{3s}.N^l$ a
N boy


S a girl praised a boy
$\text{Gcon}^{\rightarrow}_{1,2}$
$D^r.S$ praised a boy D_{3s} a girl
Conc
$D^r.S$ praised a boy
$\text{Gcon}^{\rightarrow}_{1,2}$
$D^r.S.D^l$ praised D_{3s} a boy
Conc
$D^r.S.D^l$ praised
D_{3s} a boy
$\text{Gcon}^{\leftarrow}_{1,2}$
$D_{3s}.N^l$ a N boy
Conc
$D_{3s}.N^l$ a
N boy
D_{3s} a girl
$\text{Gcon}^{\leftarrow}_{1,2}$
$D_{3s}.N^l$ a N girl
Conc
$D_{3s}.N^l$ a
N girl


S
a boy spoke to a boy and a girl
$\text{Gcon}^{\rightarrow}_{1,2}$
$D^r.S$ D_{3s}
spoke to a boy and a girl a boy
Conc
$D^r.S$
spoke to a boy and a girl
$\text{Gcon}^{\leftarrow}_{1,2}$
$D^r.S.P^l$ P
spoke to a boy and a girl
Conc
$D^r.S.P^l$
spoke
P
to a boy and a girl
$\text{Gcon}^{\leftarrow}_{1,2}$
$P.D^l$ D
to a boy and a girl
Conc
$P.D^l$
to
D
a boy and a girl
$\text{Gcon}^{\rightarrow}_{1,2}$
$D^r.D$ D_{3s}
and a girl a boy
Conc
$D^r.D$
and a girl
$\text{Gcon}^{\leftarrow}_{1,2}$
$D^r.D.D^l$ D_{3s}
and a girl
Conc
$D^r.D.D^l$
and
D_{3s}
a girl
$\text{Gcon}^{\leftarrow}_{1,2}$
$D_{3s}.N^l$ N
a girl
Conc
$D_{3s}.N^l$
a
N
girl
D_{3s}
a boy
$\text{Gcon}^{\leftarrow}_{1,2}$
$D_{3s}.N^l$ N
a boy
Conc
$D_{3s}.N^l$
a
N
boy
D_{3s}
a boy
$\text{Gcon}^{\leftarrow}_{1,2}$
$D_{3s}.N^l$ N
a boy
Conc
$D_{3s}.N^l$
a
N
boy

Of course, we still have trees with a maximum of two branches, so the binary normal form of m-MCFG is still well adapted. The property that derivation do not depend on the strings but only the types is the echo of the same property with categories in (h)MGs. Then, considering that (Conc) can only apply to two proper tuples creating a *a priori* non proper tuple, and that (Gcon) applies only to a non-proper tuple, the global operation of “fusion” between two proper tuples is the successive application of (Conc) and the closure of the resulting tuple with (Gcon). If the result is proper, then we can add the new tuple to the set of proper tuples. Otherwise, the path was a dead-end and we can forget all the preceeding operations.

One of the main remarks to do here is that these derivations are absolutely not unique. In fact, many inputs are highly ambiguous (meaning that there often exist a lot of derivations for the same sentence), and it is important for us to keep all of them. It will be important when we will implement the (Gcon) closure, because there will be sometimes many ways to do that.

We can also notice that the order in the tuple is not important, and because there can be only one occurrence of each atom in a proper tuple, proper tuples are more or less sets. Instead of using sets (which are not well suited when the tuples are not proper anymore), we will prefer keeping the atoms sorted in the tuple. This will lead to a great increase of speed in the final program, and a huge decrease of needed space to compute the conversion, thanks to many collisions. But that implies to keep record of the permutations done while sorting the tuple resulting of a (Conc).

Converting a mTPG to a MCFG is therefore also made during the closure of the mTPG lexicon with “fusion”:

(Conc)

$$\begin{pmatrix} t_1 & \dots & t_{k-1} \\ s_1 & \dots & s_{k-1} \end{pmatrix} \bullet \begin{pmatrix} t_k & \dots & t_n \\ s_k & \dots & s_n \end{pmatrix} = \begin{pmatrix} t_1 & \dots & t_n \\ s_1 & \dots & s_n \end{pmatrix}$$

$$\Downarrow$$

$$\sigma(\tau(t_1 \dots t_n)) \to \sigma(t_1 \dots t_{k-1}) \quad \sigma(t_k \dots t_n) \quad \Pi(t_1 \dots t_{k-1}, t_k \dots t_n)$$

Where τ is the sorting function and Π the correct printing of the permutation associated to τ. We assume here that the sequences $t_1 \dots t_{k-1}$ and $t_k \dots t_n$ are already sorted.

$\textbf{(Gcon)}^{\leftarrow}_{i,j}$ [1]

$$\begin{pmatrix} t_1 & \dots & t_i & \dots & t_j & \dots & t_n \\ s_1 & \dots & s_i & \dots & s_j & \dots & s_n \end{pmatrix} \to \begin{pmatrix} t_1 & \dots & t'_i & \dots & t_n \\ s_1 & \dots & s_i s_j & \dots & s_n \end{pmatrix}$$

$$\Downarrow$$

$$\sigma(t_1 \dots t_i \dots t_j \dots t_n) \to \sigma(t_1 \dots t'_i \dots t_n)$$

$$[0,0] \dots [0,i-1][0,i;0,j][0,i+1] \dots [0,j-1][0,j+1] \dots [0,n-1]$$

$\textbf{(Gcon)}^{\rightarrow}_{i,j}$

$$\begin{pmatrix} t_1 & \dots & t_i & \dots & t_j & \dots & t_n \\ s_1 & \dots & s_i & \dots & s_j & \dots & s_n \end{pmatrix} \to \begin{pmatrix} t_1 & \dots & t'_i & \dots & t_n \\ s_1 & \dots & s_j s_i & \dots & s_n \end{pmatrix}$$

$$\Downarrow$$

$$\sigma(t_1 \dots t_i \dots t_j \dots t_n) \to \sigma(t_1 \dots t'_i \dots t_n)$$

$$[0,0] \dots [0,i-1][0,j;0,i][0,i+1] \dots [0,j-1][0,j+1] \dots [0,n-1]$$

It is noticeable that the output of (Gcon) is sorted if its input is.

The last thing to take care of is the building of the lexicon, where tuples with an arity greater than 2 can be inputted while the binary normal form of m-MFCG cannot use them directly. For each tuple, we therefore create a balanced binary tree to assemble the tuple (example taken from [11]: $\begin{pmatrix} v & D^r.s & R^r.o \\ \text{praises} & \text{nom} & \text{acc} \end{pmatrix}$):

---

[1] for reading convenience and conciseness, i is put before j in the tuples, but it is not required in the general case

$$\begin{array}{rcl}
\alpha_0 & \rightarrow & \text{"praises"} \\
\alpha_1 & \rightarrow & \text{"nom"} \\
\alpha_2 & \rightarrow & \text{"acc"} \\
\alpha_3 & \rightarrow & \alpha_0 \quad \alpha_1 \quad [0,0][1,0] \\
\alpha_4 & \rightarrow & \alpha_3 \quad \alpha_2 \quad [0,0][0,1][1,0] \\
\sigma((v, D^r.s, R^r.o)) & \rightarrow & \alpha_4 \quad [0,0][0,1][0,2]
\end{array}$$

Where the α_i are symbols uniquely associated to the given tuple (In particular, they are not in the range of σ)

Applying the closure of (Conc) and (Gcon) on the grammar given above as an example, we obtain:

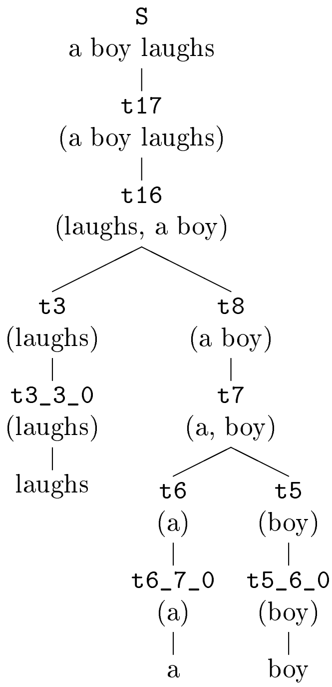
```
S --> t17 [0,0]
t17 --> t33 [0,0;0,1]
t33 --> t15 t26 [1,0][0,0]
t15 --> t30 [0,1;0,0]
t30 --> t2 t26 [1,0][0,0]
t2 --> t2_2_0 [0,0]
t2_2_0 --> "praised"
t26 --> t35 [0,0;0,1]
t35 --> t20 t26 [1,0][0,0]
t20 --> t31 [0,1;0,0]
t31 --> t4 t26 [1,0][0,0]
t4 --> t4_4_0 [0,0]
t4_4_0 --> "and"
t26 --> t34 [0,1;0,0]
t34 --> t19 t26 [1,0][0,0]
t19 --> t31 [0,0;0,1]
t19 --> t18 [0,1;0,0]
t18 --> t4 t8 [0,0][1,0]
t8 --> t7 [0,0;0,1]
t7 --> t6 t5 [0,0][1,0]
t6 --> t6_7_0 [0,0]
t6_7_0 --> "a"
t5 --> t5_6_0 [0,0]
t5_6_0 --> "boy"
t5 --> t5_5_0 [0,0]
t5_5_0 --> "girl"
t26 --> t27 [0,1;0,0]
t27 --> t20 t8 [0,0][1,0]
t20 --> t18 [0,0;0,1]
t26 --> t25 [0,0;0,1]
t25 --> t19 t8 [0,0][1,0]
t15 --> t22 [0,1;0,0]
t22 --> t1 t10 [1,0][0,0]
t1 --> t1_1_0 [0,0]
t1_1_0 --> "spoke"
t10 --> t28 [0,1;0,0]
t28 --> t0 t26 [1,0][0,0]
t0 --> t0_0_0 [0,0]
t0_0_0 --> "to"
t10 --> t9 [0,0;0,1]
t9 --> t0 t8 [0,0][1,0]
t15 --> t13 [0,0;0,1]
t13 --> t2 t8 [0,0][1,0]
t17 --> t32 [0,1;0,0]
t32 --> t14 t26 [1,0][0,0]
t14 --> t30 [0,0;0,1]
t14 --> t13 [0,1;0,0]
t17 --> t24 [0,1;0,0]
t24 --> t15 t8 [0,0][1,0]
t17 --> t23 [0,0;0,1]
```

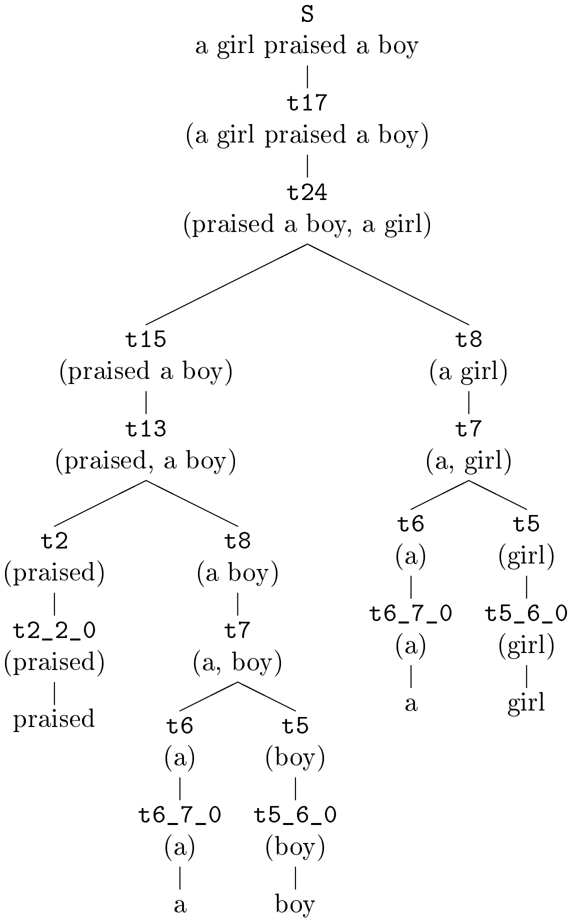
```
t23 --> t14 t8 [0,0][1,0]
t17 --> t21 [0,1;0,0]
t21 --> t12 t10 [1,0][0,0]
t12 --> t29 [0,0;0,1]
t29 --> t1 t26 [1,0][0,0]
t12 --> t11 [0,1;0,0]
t11 --> t1 t8 [0,0][1,0]
t17 --> t16 [0,1;0,0]
t16 --> t3 t8 [0,0][1,0]
t3 --> t3_3_0 [0,0]
t3_3_0 --> "laughs"
```

With these correspondances:

```
t0 : (P.D‘)
t1 : (D’.S.P‘)
t2 : (D’.S.D‘)
t3 : (D3s’.S)
t4 : (D’.D.D‘)
t5 : (N)
t6 : (D3s.N‘)
t7 : (D3s.N‘;N)
t8 : (D3s)
t9 : (P.D‘;D3s)
t10 : (P)
t11 : (D’.S.P‘;D3s)
t12 : (S.P‘)
t13 : (D’.S.D‘;D3s)
t14 : (S.D‘)
t15 : (D’.S)
t16 : (D3s’.S;D3s)
t17 : (S)
t18 : (D’.D.D‘;D3s)
t19 : (D.D‘)
t20 : (D’.D)
t21 : (P;S.P‘)
t22 : (P;D’.S.P‘)
t23 : (S.D‘;D3s)
t24 : (D’.S;D3s)
t25 : (D.D‘;D3s)
t26 : (D)
t27 : (D’.D;D3s)
t28 : (D;P.D‘)
t29 : (D;D’.S.P‘)
t30 : (D;D’.S.D‘)
t31 : (D;D’.D.D‘)
t32 : (D;S.D‘)
t33 : (D;D’.S)
t34 : (D;D.D‘)
t35 : (D;D’.D)
```

Derivations of the sentences “a boy laughs” and “a girl praised a boy” in that grammar are:


S
a boy laughs
t17
(a boy laughs)
t16
(laughs, a boy)
t3
(laughs)
t3_3_0
(laughs)
laughs
t8
(a boy)
t7
(a, boy)
t6
(a)
t6_7_0
(a)
a
t5
(boy)
t5_6_0
(boy)
boy


S
a girl praised a boy
t17
(a girl praised a boy)
t24
(praised a boy, a girl)
t15
(praised a boy)
t13
(praised, a boy)
t2
(praised)
t2_2_0
(praised)
praised
t8
(a boy)
t7
(a, boy)
t6
(a)
t6_7_0
(a)
a
t5
(boy)
t5_6_0
(boy)
boy
t8
(a girl)
t7
(a, girl)
t6
(a)
t6_7_0
(a)
a
t5
(girl)
t5_6_0
(girl)
girl

4 Implementations

4.1 mg2mcfg

Another aspect of the limitations of the parser is that it is not possible to recognize idioms. But it is very interesting for us to be able to have something like (example taken from the file `examples/g10.mg` distributed with the program `mg2mcfg`):

```
"it is not the case that" :: =v v
```

As we did in section 3.3 with tuples, we are going to build a balanced binary tree but now merging strings:

$$\begin{array}{rcl}
\alpha_1 & \rightarrow & \text{"it"} \\
\alpha_2 & \rightarrow & \text{"is"} \\
\alpha_3 & \rightarrow & \text{"not"} \\
\alpha_4 & \rightarrow & \text{"the"} \\
\alpha_5 & \rightarrow & \text{"case"} \\
\alpha_6 & \rightarrow & \text{"that"} \\
\alpha_7 & \rightarrow & \alpha_1 \quad \alpha_2 \quad [0,0;1,0] \\
\alpha_8 & \rightarrow & \alpha_3 \quad \alpha_4 \quad [0,0;1,0] \\
\alpha_9 & \rightarrow & \alpha_5 \quad \alpha_6 \quad [0,0;1,0] \\
\alpha_{10} & \rightarrow & \alpha_7 \quad \alpha_8 \quad [0,0;1,0] \\
\alpha_{11} & \rightarrow & \alpha_{10} \quad \alpha_9 \quad [0,0;1,0] \\
\sigma(= v\ v) & \rightarrow & \alpha_{11} \quad [0,0]
\end{array}$$

Where the α_1 do not occur anywhere else.

From now on, we know that we need to compute the closure of a given MG lexicon with the merge and move functions, but we have not yet discussed how. We will use the classical chart/agenda scheme:

- a dynamic array of lists `cat` containing (with no duplicates) tuples where the first chain is a simple feature (a category) with no licensee.
- a dynamic array of lists `cat_move` containing (with no duplicates) tuples where the first chain has a category as first feature and licensees behind.
- a dynamic array of lists `sel` containing (with no duplicates) tuples where the first feature of the first chain is a selecting feature.
- a dynamic array of lists `lic` containing (with no duplicates) tuples where the first feature of the first chain is a licensor.

These tuples are stored at the index equal to the code corresponding to the string of the first feature (we will call this integer the "key" of the tuple). Therefore, it makes it very easy to know which operations are possible: if the category D has $k \in \mathbb{N}$ for a coding, elements like `Mary :: D`, `who :: D -wh` and `likes :: =D =D V` are stored in the list in the $(k+1)$-th cell of their respective array (`cat`, `cat_move` and `sel`).

We also have a function for inserting in the right array the result of a merge or move function (which we use at the beginning with the lexicon). Before inserting, we look if the array contained the element, so that we don't insert elements twice. If the element is new, its key is also added to an integer set used as a agenda. After every current possible function is applied, we look at the agenda and if it is nonempty, we re-apply merges et moves only for the concerned keys.

The "dynamic arrays" mentioned above are implemented in a specific module (`Dyn_array`), making it easier to share with the two other programs. They are arrays of lists of a given `'a` type and a counter p used to know how many elements the lists contain. The size of the array grows with insertions: inserting an element in cell k when the array has size $n < k$ resizes the array to size $\min(2k+1, \mathrm{Sys.max\_array\_length})$. Retrieving a list of elements having the same key is performed in constant time, finding a particular element is $O(p)$ and so is insertion because we don't want duplicates. Note that p complexity occurs only if all the elements have the same key and that the array has size 1, which is far for the general case.

Attributing unique integers to various types of objects is another common issue in the three programs. For that, I have created another module, but using a functor `Dict.Make` (thus adapting to all the objects we need). The main requirement about the input module for the functor is that the main type should be comparable, and contain that comparison. What I called a "dictionary" is composed of a balanced binary tree used as a Map, an array of variable length and a counter p. Starting from an empty Map and array, and a counter set to (-1), when we try to know which integer is associated to a given object, we find out from the Map if an integer has already been assigned to it. If not, we increase the counter, add the (object,integer) pair in the map, and the object in the cell numbered by the integer in the array (doubling its size if necessary). Therefore, the "object $\mapsto$ integer" translation is $O(p + \ln p)$ while the "integer $\mapsto$ object" one is made in constant time. Note that the p is here only for taking consideration of array resizing (which only occur every $2^n + 1$ new objets), so the function is usually significantly faster. Of course, space complexity is not optimal, but it is still $O(p)$.

Here are the objects that needed a "dictionary":

- Features (the keys for the tuples)
- Tuples (hence defining σ and the symbols of the mcfg)
- Left parts of mcfg rules (keys for right parts)
- Right parts of mcfg rules

We also use a `Dyn_array` for mcfg rules, right parts being indexed by left parts of the rules. Combined with a function associating to a right part the possible left parts (for example and schematicaly, if LEFT(a) –> RIGHT(b c <map>) is a rule, then applied to that function, { LEFT(b), LEFT(c) } would be the result), it makes it very efficient to filter the final mcfg collection for accessibility from the start rule S (co-accessibility is assured in a natural manner by contruction of that collection, the closure being operated "from the leaves to the root"). That operation also gives us the accessible tuples, and that information is interesting since we want to produce a file containing correspondance between mcfg symbols and tuples as to verify and analyze mcfg derivation trees more easily.

Correctness is assured by the theorems proving weak equivalency between these grammars, in addition to the straight-forward proves that each mcfg rules (and in particular the map) corresponds to the application of the given function.

We can also establish that the program terminates: being given a finite lexicon, the number of different features is finite, entailing that the maximum arity of a tuple is (you cannot have a tuple with the same first feature at two different coordinates). Moreover, the length of the sequences of licensees in the lexicon is bound, and rules cannot increase them. Finally, every function (i.e merge_1, merge_2, merge_3, move_1 and move_2) makes the expression of the first coordinate of a tuple decrease exactly of one (move even reduces the size of the tuple). The set of possible tuples entailed by any finite lexicon is therefore finite, and the closure of this lexicon is included in that set. So in the main loop of the closure, there exists a step when nothing changes anymore, and the program terminates.[2]

For commidity, I added using *ocamllex* and *ocamlyacc* a lexer and a parser in order to facilitate the input of MG. Here are the definitions:

mg → *startlist lexrulelist ENDGRAMMAR*
startlist → *IDENT slist ENDRULE*

---

[2] space and time complexity are more complex questions, and I have not had the time to solve them yet

slist → ∅ | *IDENT slist*
lexrulelist → ∅ | *lexrule lexrulelist*
lexrule → *vocab sign featurelist category licenseelist ENDRULE*
sign → *DEFINE*
vocab → *IDENT*
featurelist → ∅ | *feat featurelist*
feat → *featmod IDENT*
featmod → *EQUAL* | *PLUS*
category → *IDENT*
licenseelist → ∅ | *licensee licenseelist*
licensee → *MINUS IDENT*

where the lexemes (ENDGRAMMAR, IDENT, ENDRULE, DEFINE, EQUAL, PLUS, MINUS) are defined mglexer.mll:

```
{
(** Returns a stream of lexemes defined in mglexer.mll
@author Matthieu Guillaumin
@see ’mglexer.mll’ for lexemes used *)
open Mgparser;;
exception LexingError of int*string
}

rule token = parse
 [’ ’ ’\t’ ’\n’ ’\r’] {token lexbuf}
| ( ’/’ [^’/’]* ’/’)   {token lexbuf}      (* comments are between ’/’s *)
| ( ’%’ [^’\n’]* ’\n’ )   {token lexbuf}   (* or behind a % *)
| ’;’ {ENDRULE}
| "::" {DEFINE}
| ’=’ {EQUAL}
| ’+’ {PLUS}
| ’-’ {MINUS}
| ([’a’-’z’] | [’A’-’Z’] | [’0’-’9’])+ {IDENT(Lexing.lexeme lexbuf)}
| (’\"’ ([^’\"’]* as idiom) ’\"’) {IDENT(idiom)}
| eof {ENDGRAMMAR}
| (_) {raise(LexingError(Lexing.lexeme_start lexbuf,Lexing.lexeme lexbuf))}
```

Some additional verifications about basic properties are made with semantic actions, and warnings are printed (with rule number) when needed:

- The starting category symbols are used
- The first feature of a chain is not a licensor
- A selecting feature corresponds to a defined category
- A licensee has a defined corresponding licensor
- A licensor has a defined corresponding licensee
- A category is a starting category or is selected

The second one could have been easily implemented in the grammar, but as it is a common mistake, a warning giving the location of the error is useful.

4.2 hmg2mcfg

Implementing hMG is basically the same as MG. We only do minor changes for lexical terms and the starting rule, and of course we increase the number of functions, types of rules and `Dyn_array` we use (because we have also more types of features):

- mcfg rules
- selectors
- categories
- categories with licensee features
- licensors
- right incorporators (head movements on the right)
- left incorporators
- right affix hopping
- left affix hopping

Finally, we need a way of dealing with relations for adjunction. Adjunctions are peculiar because they are a sort of meta-rules. We create two sets for respectively left and right adjunctions, we also have the functions left-adjoin1, right-adjoin1, left-adjoin2 and right-adjoin2. At the end of each loop in the closure, we look for new elements that are concerned with an adjunction and apply them if every complies.

The new types of features and the adjunctions lead us to modify the lexer and the parser. The changes are minor though. In order to take advantage of the numerous example files written for a prolog parser for hMG, I added a new set of lexer/parser for those files. Finally, we have to change the verifications made, the new ones are:

- The starting category symbols are used
- The first feature of a chain is not a licensor
- A selecting feature corresponds to a defined category
- A licensee has a defined corresponding licensor
- A licensor has a defined corresponding licensee
- A category is a starting category, is selected or is an adjoin to a defined category
- The categories involved in an adjunction are defined

The correctness and the termination of this program is basically the same than for mg2mcfg, there is just more cases due to additionnal functions.

4.3 mtpg2mcfg

We also have a lexer and a parser for this program:

mtpg → *startlist orderlist lexrulelist ENDGRAMMAR*
startlist → *IDENT slist*
slist → ∅ | *slist*
orderlist → ∅ | *BEGINORDER IDENT SMALLER IDENT ENDORDER orderlist*
| *BEGIN ORDER IDENT GREATER IDENT ENDORDER orderlist*
lexrulelist → ∅ | *BEGINRULE lexrule ENDRULE lexrulelist*

lexrule → *couple* | *couple SEMICOLON lexrule*
couple → *typed COMA IDENT* | *typed COMA*
typed → *IDENT mode* | *IDENT mode DOT typed*
mode → $\emptyset$ | *LEFTINVERSE* | *RIGHTINVERSE*

Where the lexer is:

```
{
(** Returns a stream of lexemes defined in mtpglexer.mll
@author Matthieu Guillaumin
@see ’mtpglexer.mll’ for lexemes used *)
open Mtpgparser;;
exception LexingError of int*string
}

rule token = parse
 [’ ’ ’\t’ ’\n’ ’\r’] {token lexbuf}
| ( ’/’ [^’/’]* ’/’ ) {token lexbuf} (* ignoring things between ’/’s *)
| ( ’%’ [^’\n’]* ’\n’ )   {token lexbuf} (* everything behind a % is ignored *)
| ’)’ {ENDRULE}
| ’(’ {BEGINRULE}
| ’[’ {BEGINORDER}
| ’]’ {ENDORDER}
| ’<’ {SMALLER}
| ’>’ {GREATER}
| ’.’ {DOT}
| ’,’ {COMA}
| ’;’ {SEMICOLON}
| ’\’’ {RIGHTINVERSE}
| ’‘’ {LEFTINVERSE}
| ([’a’-’z’ ’A’-’Z’ ’0’-’9’])+ {IDENT(Lexing.lexeme lexbuf)}
| (’\"’ ([^’\"’ ’;’]* as idiom) ’\"’) {IDENT(idiom)}
| eof {ENDGRAMMAR}
| (_) {raise(LexingError(Lexing.lexeme_start lexbuf,Lexing.lexeme lexbuf))}
```

The basic verifications on the output of the parser are:

- The types are well formed (right inverses, atom, left inverses)
- The atoms are unique in a tuple
- The inverses correspond to an atom
- The right (resp. left) part of a $<$ (resp. $>$) relation is a start symbol or correspond to an inverse
- The atoms are start symbols or defined inverses or left (resp. right) part of a $<$ (resp. $>$) relation
- The start symbols are defined atoms or a right (resp. left) part of a $<$ (resp. $>$) relation

As we mentioned earlier, the order of typed strings is not important inside a tuple. In order to limit the number of different possible tuples, keeping them sorted (numerically by the atom's corresponding integer) is of great benefit. First this property is conserved when applying (Gcon). Let's study the effect with (Conc). Of course, when we concatenate two tuples, the result is not sorted in general (like the inputted tuples for which we need a full merge sort), so instead of concatenating them, we apply a slightly modified single-step merge sorting (because we know that the two tuples are sorted). It is modified because we need to keep track of the permutation σ associated to this sorting, in order to take it in consideration in the map function of the mcfg rule: We begin by giving to each element in the tuple (of size n) its coordinate (that is done in $O(n)$). Then we apply the merge step ($O(n)$). Finally, we split ($O(n)$ again, so is the global operation) the resulting list in two separate lists containing respectively the sorted elements and the list of indexes of these elements (therefore, this list contains a representation of σ). For example:

$$\begin{pmatrix} a & b & d & f \\ 0 & 1 & 2 & 3 \end{pmatrix} \quad \begin{pmatrix} c & e \\ 4 & 5 \end{pmatrix}$$

$$\downarrow \text{(Conc)}$$

$$\begin{pmatrix} a & b & c & d & e & f \\ 0 & 1 & 4 & 2 & 5 & 3 \end{pmatrix}$$

We will apply this permutation when printing the mcfg rule:

$$\begin{pmatrix} a & b & c & d & e & f \end{pmatrix} \rightarrow \begin{pmatrix} a & b & d & f \end{pmatrix} \quad \begin{pmatrix} c & e \end{pmatrix} \quad [0,0][0,1][1,0][0,2][1,1][0,3]$$

Of course, we will try to narrow our efforts to only applying the (Conc)+(Gcon)-closure scheme to two tuples that may at the end produce at least one proper tuple. Consider the following necessary condition:

Lemma 4.1 *If two tuples t_1 and t_2 produce a proper tuple after (Conc) and a (Gcon) closure, then for any element a in the intersection of the sets of their atoms, there is an inverse of type b for that a (i.e such as $b \geq a$) in at least one tuple between t_1 and t_2.*

The proof is straight forward, because if we obtain a proper tuple, there are no more duplicate atom, so if the intersection of the sets of the atoms of the tuples was nonempty, at least one occurrence of each duplicate atom has disappeared, and that is possible only if a corresponding inverse (formally any inverse of type b such as $b \geq a$ or equivalently we can say that any c such as $c \leq b$ has a corresponding inverse in the location of b) is present. For the implementation, the fact that every tuple is sorted makes the process of computing the intersection of the atoms more efficient ($O(n_1 + n_2)$) than the general case.

Now, how are the (Gcon) function and the (Gcon) closure implemented? The function itself is quite easy: knowing the indexes i and j of the concerned types we just need to remove the $j-$th element from the tuple and remove the good adjoint-type appearing in the $i-$th coordinate. (Gcon) closure is more tricky. We first need a function called `analyze` which will return *every* possible association between a saturated type and an adjoint-type on the edge of a type in a given tuple. In that operation, we need to keep the respective coordinates of the inverses and their types. The output is made of two lists of same size: one containing couples of a saturated atom and its position, the other containing the lists of complying inverses (in respect of the relation) with their position and type.

With these functions, we can obtain a recursive definition of (Gcon) closure (`tr` is a function of translation, `lower` returns the list of elements smaller than the given element, `assoc_all` is a general function on association lists returning *all* the elements associated to the given one, `gcon` is the (Gcon) function, `complies_atom_unicity` is a function deciding if a tuple is proper, `unicize` is a function removing any duplicate element in list):

```
let rec gcon_closure tr lower (t,n,l) =
  let (sat,unsat) = analyze lower 0 [] [] t
  in let sat_dest = List.map (function (x,_) -> assoc_all x unsat) sat
  in let possible_gcons = List.flatten
                  (List.map2 (fun (_,r) s -> List.map (function (p,q) -> (p,r,q) ) s) sat sat_dest)
  in let applied_gcons = List.map (function (x,y,z) -> gcon tr x y z (t,n,l)) possible_gcons
  in
    match applied_gcons with
```

```
    | [] -> (if complies_atom_unicity t then [(t,n,l)] else []),[]
    | l  -> (let tuple_list,mcfg_list = List.split l
            in let tuples,rules = List.split (List.map (gcon_closure tr lower) tuple_list)
            in let full_list = List.combine tuples (List.map2 (fun a li -> a::li) mcfg_list rules)
            in let good_list = List.filter (fun (x,_) -> x<>[]) full_list
            in let new_t,new_r = List.split good_list
            in unicize(List.flatten new_t), List.flatten new_r)
```

While keeping only the final results of the closure, we have in the mcfg rule list every separate step that leads to each result.

Because the problem of closing a mTPG lexicon is in a certain sense a problem of more complexity then for (h)MG, we will also have to be more subtle, and for example mark the difference between tuples computed in the current loop and older ones. The data structures are:

- "Dictionaries" for types and tuples
- Two `dyn_arrays` for relations: one containing greater types, the other with smaller ones.
- Dictionaries for left and right parts of mcfg rules
- `Dyn_arrays` for all mcfg rules and another one for printed ones only
- A mutable set of integers containing old saturated tuples (their translation)
- A mutable set of integers containing new saturated tuples (the ones created in the last loop)
- A mutable temporary set of integers containing the saturated tuples created in this loop
- A `dyn_array` for old unsaturated tuples, present in every cell where they have an inverse (we do not apply relations here, it is done elsewhere)
- Two other `dyn_arrays` for new and temporary unsaturated tuples
- A global boolean variable deciding if something has changed during the loop

At each loop, after computations, we add to the old elements the new ones, we replace the "new" by the "temporary" ones and we empty the temporary sets and arrays. Now, inside a loop, we look for application of the functions in:

- new unsaturated tuples on new saturated ones
- old unsaturated tuples on new saturated ones
- new unsaturated tuples on old saturated ones

For each one between these three cases, we associate saturated tuples with unsaturated ones that have at least one compatible inverse on one "edge" of a complex type (applying relations), check with `has_a_chance` if (Conc) and (Gcon) closure are a priori worth trying and finally apply (Conc) and (Gcon) closure. Then, we add the results, if they are not in the "new" or "old" sets, to the "temporary" one, and the mcfg rules to the collection and set if needed the global boolean to "true".

As for mTPG, correctness is a consequence of the proof of weak-equivalency and termination is assured by the finiteness of the set of possible tuples: indeed, there is only a finite number of simple types, so a finite number of different atoms, and the arity of tuples is bound. And because operations can only decrease the number of inverses on the left or on the right of each atom, the total number of possible proper tuples defined from a given finite lexicon is finite. And we compute only a subset of that. Arrived at a certain point, there is no more new tuples that can be created, so the program ends.

5 Comparison with existing parsers

5.1 MG and hMG

Before comparing the parsing between the different existing parsers for (h)MGs and the MCFG parser after using my programs, we should look at the time spend by them.[3] Unfortunately, tests have not been executed on a large scale, but on only 11 typical grammars for mg2mcfg and approximatively 20 for hmg2mcfg. The biggest and slowest of all is the larsonian grammar (in file `hmg2mcfg/examples/larsonian1.pl`), and the computation takes about 1 second.

The existing parsers for (h)MG are:

- a CKY MG parser for SWI prolog (Stabler, UCLA)
- a CKY MG parser for SICSTUS prolog (Stabler, UCLA)
- a CKY MG parser written in Scheme (Nigoyi, MIT)
- a CKY MG parser written in OCaml (Hale, MSU) [4]

And here are the results, due to Stabler, with the larsonian grammar (`hmg2mcfg/examples/larsonian.pl`) and on the same machine:

- "they have -ed forget -en that the boy who tell -ed the story be -s so young"

Program	milliseconds	chart size
mgcky-swi	1,229,600	11559
mgcky-sicstus	1,530,630	11559
mgcky-ocaml (Hale)	75,810	11297
mg-mcfg (Guillaumin, Albro)	293	1317

- "the fact that the girl who pay -ed for the ticket -s be -s very poor doesnt matter"

Program	milliseconds	chart size
mgcky-sicstus	440,310	6466
mgcky-ocaml (Hale)	27,440	5913
mg-mcfg (Guillaumin, Albro)	597	1306

There are many other examples, but the two extremes are:

- "he remember -ed that the food which Chris pay -ed the bill for be -ed cheap"

Program	milliseconds
mgcky-sicstus	44,020
mgcky-ocaml (Hale)	5,600
mg-mcfg (Guillaumin, Albro)	199

and finally:

- "he remember -ed that the sweet -s which David give -ed Sally be -ed a treat"

Program	milliseconds
mgcky-sicstus	234,101,050
mgcky-ocaml (Hale)	2,613,570
mg-mcfg (Guillaumin, Albro)	646

---

[3] This is not fundamental, though, because the translation between (h)MG and MCFG has to be done only once, for a possibly infinite number of parsing

[4] all references on `http://www.humnet.ucla.edu/humnet/linguistics/people/stabler/epssw.htm`

5.2 mTPG

Again, we have to see if running the program is acceptable, and again, we only have few (6) relevant examples to try. For 4 of them, it takes less than a tenth of a second. For the fifth (`mtpg2mcfg/examples/g4.mtpg`), it takes about 12 seconds (on my personnal computer), and the sixth (`mtpg2mcfg/examples/g4wh.mtpg`, where wh-movements are added to the former grammar) takes about half an hour but consumes a lot of memory, which can be problematic.

The existing parsers for mTPG are:

- a CKY mTPG parser for SICSTUS prolog (Stabler, UCLA)
- a CKY mTPG parser for SWI prolog (Stabler, UCLA) [5]

The results using a simple grammar (`mtpg2mcfg/examples/g0wg.mtpg` and `tpgcky-swi/grammars/g1.pl`) on the same machine are, for example:

- “the girl who praised a boy who laughs spoke to a boy who praised a girl”

Program	milliseconds	chart size
tpgcky-swi	232,230	3360
mtpg-mcfg (Guillaumin, Albro)	48	1567

Acknowledgments

I would like to thank Pr Edward Stabler, for his thoughts and always very enlightening views on the problems raised during my internship, and on language complexity in general, and for his availability and enthusiasm; Pr Dominique Sportiche, for introducing me to the field of linguistics, for making this internship possible and for his concern; and the entire Department of Linguistics at UCLA for having made my stay in Los Angeles very enjoyable and intellectually rich.

---

[5] all references on `http://www.humnet.ucla.edu/humnet/linguistics/people/stabler/epssw.htm`

References

[1] Daniel M. Albro. An earley-style recognition algorithm for mcfgs. Not published, 2000.

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

[3] Hendrik Harkema. *Parsing Minimalist Languages.* PhD thesis, University of California, Los Angeles, 2001.

[4] Edward Stabler Hilda Koopman, Dominique Sportiche. An introduction to syntactic analysis and theory. University of California, Los Angeles.

[5] A. K. Joshi, K. Vijay-Shanker, and D. Weir. The convergence of midly context-sensitive grammars. In S. M. Shieber and T. Wasow, editors, *The Processing of Natural Language Structure*, pages 999–999. MIT Press, 1992.

[6] Jens Michaelis. Derivational minimalism is mildly context-sensitive. *Lecture Notes in Computer Science*, 2014:179–??, 2001.

[7] H. Seki, T. Matsumura, M. Fujii, and T. Kasami. On multiple context-free grammars. *Theoretical Computer Science*, 88(2):191–229, October 1991.

[8] Edward Stabler. Derivational minimalism. In Christian Retoré, editor, *Proceedings of the 1st International Conference on Logical Aspects of Computational Linguistics (LACL-96)*, volume 1328 of *LNAI*, pages 68–95, Berlin, September 23–25 1997. Springer.

[9] Edward P. Stabler. Recognizing head movement. *Lecture Notes in Computer Science*, 2099:245–??, 2001.

[10] Edward P. Stabler. *Notes on computational linguistics*, chapter 9 and 10. 2003. University of California, Los Angeles.

[11] Edward P. Stabler. Tupled pregroup grammars. University of California, Los Angeles, 2004.