Mildly Context Sensitive Grammar Formalisms

Petra Schmidt

Pfarrer-Lauer-Strasse 23
66386 St. Ingbert
petra.schmidt@edeka-suedwest.de

Overview

Introduction

Natural languages are not context free. An example non-context-free phenomenon are cross-serial dependencies in Dutch, an example of which we show in **Fig. 1.**

ik haar hem de nijlpaarden zag helpen voeren

Fig. 1.: cross-serial dependencies

A concept motivated by the intention of characterizing a class of formal grammars, which allow the description of natural languages such as Dutch with its cross-serial dependencies was introduced in 1985 by Aravind Joshi. This class of formal grammars is known as the class of mildly context-sensitive grammar-formalisms.

According to Joshi (1985, p.225), a mildly context-sensitive language L has to fulfill three criteria to be understood as a rough characterization. These are:
- The parsing problem for L is solvable in polynomial time
- L has the constant growth property
- There is a finite figure n, that limits the maximum number of instantiations of cross serial dependencies occurring in a sentence of L

A collection of mildly context-sensitive grammars formalisms is given by Joshi et al. (1991):
- Tree Adjoining Grammars (TAGs)
- Combinatory Categorial Grammars (CCGs)
- Linear Indexed Grammars (LIGs)
- Head Grammars (HGs)
- Multicomponent TAGs (MCTAGs)
- Linear Context-Free Rewriting Systems (LCFRSs)

According to the string-languages they generate TAGs, CCGs, LIGs and HGs are equivalent. MCTAGs and LCFRSs subsume TAGs, CCGs, LIGs and HGs.

In this paper I will concentrate on the grammar formalisms of TAG, CCG and LIG. I will present TAG, CCG, and LIG and further the proof of Vijay-Shanker and Weir, which shows that they are equivalent with respect to the string-languages they describe. In computational linguistics, TAG and CCG are the formalisms that are really used, LIG is used in a more theoretical way, e.g. for proofs and for parsing-algorithms. I will conclude by presenting LCFRS and MCFG by example.

Tree Adjoining Grammars (TAGs)

Tree Adjoining Grammars were developed by Aravind Joshi in the seventies and eighties. They consist of initial and auxiliary trees that can be combined by the operations of adjunction and substitution. The union of initial and auxiliary trees is called the elementary trees.

Definition

A tree-adjoining grammar (TAG) G is a quintuple $\{V_N, V_T, T_{ini}, T_{aux}, S\}$, where
V_N is a finite set of non-terminals
V_T is a finite set of terminals
T_{ini} is a finite set of labled trees, called the initial trees
T_{aux} is a finite set of labled trees, called the auxiliary trees
and $S \in V_N$ is the start symbol.

Initial trees are of the form as shown in **Fig. 2.**

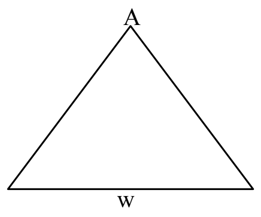


Fig. 2.: initial tree, with root labeled with non-terminal A and yield labeled w, where w is a series of terminals and substitution-nodes.

Auxiliary trees are of the form as shown in **Fig. 3.**

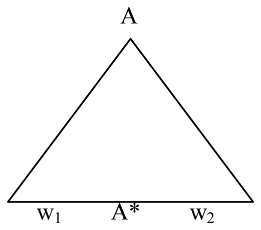


Fig.3.: auxiliary tree, with root labeled with non-terminal A and yield labeled $w_1A^*w_2$, where w_1 and w_2 are series of terminals and substitution-nodes and A* is a non-terminal.

Trees can be derived from others by applying the structure building operations of substitution and adjunction.

Definition substitution

To replace a leaf of a labeled tree ϕ with label $A\downarrow$, where $A\downarrow$ is a substitution node, by an initial tree ψ with root labeled A is a structure building operation called substitution. (see **Fig. 4.**)

Definition adjunction

Let ϕ be a labeled elementary tree, which contains a node labeled A for some $A \in V_N$. and ψ is a labeled auxiliary tree with root labeled A and it has a leaf labelled A*. The structure-building operation adjunction replaces the part of ϕ below the node A by the labeled tree ψ and substitutes the node A* of ψ by the replaced part of ϕ.
(see **Fig. 5.**)

The operation substitution can be modeled by the operation adjunction. In that case the node A of the elementary tree ϕ is on the yield of ϕ and there is a tree which can derive the A* of ψ to ε.

Definition string language L(G)

The string language that is constructed by a TAG $G = \{V_N, V_T, T_{ini}, T_{aux}, S\}$ is the closure of T_{ini} and T_{aux} under finitely many applications of substitution and adjunction.

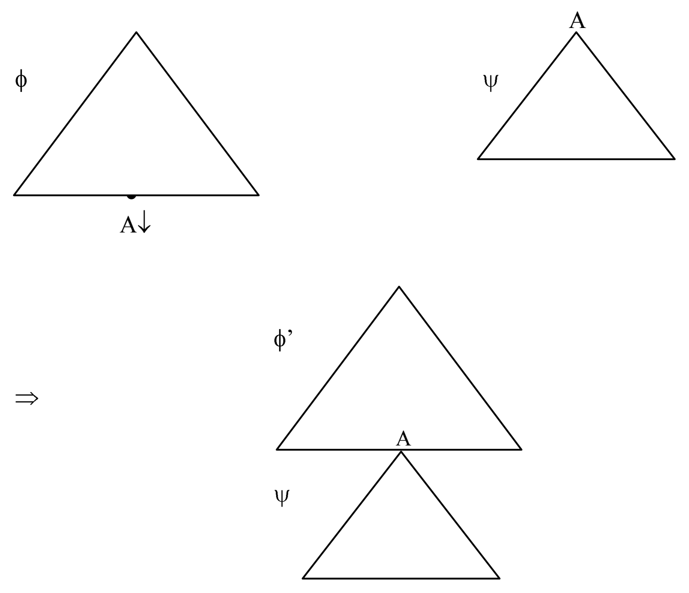


Fig. 4.: Substitution

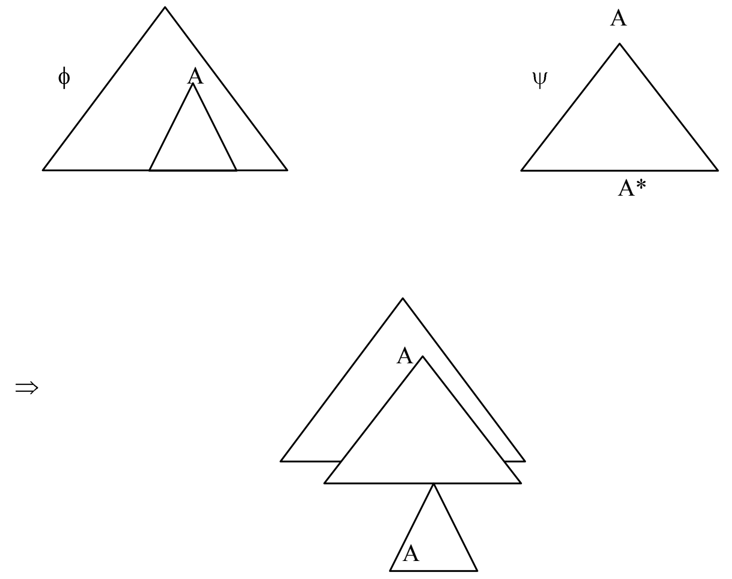
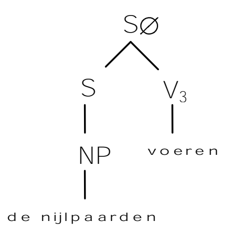
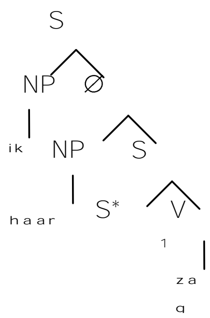


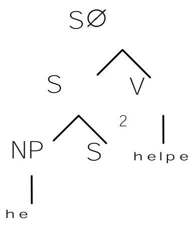
Fig. 5.: Adjunction

For example see the tree-adjoining-grammar for the dutch sentence: “ik haar hem de nijlpaarden zag helpen voeren”:

G = {V_N, V_T, T_{ini}, T_{aux}, S}, where
V_N = {S, NP, V1, V2, V3}
V_T = {ik, haar, hem, de nijlpaarden, helpen voeren}
T_{ini} ={α_1}
T_{aux} = {β_1, β_2}
S ∈ V_N start symbol

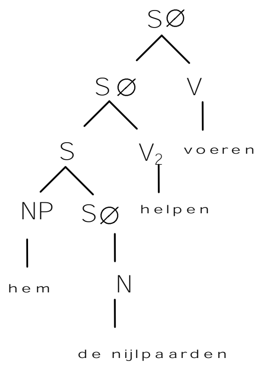






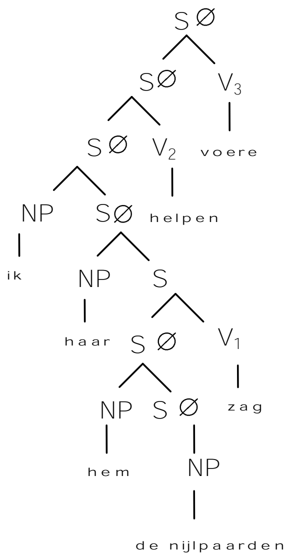
α₁: initial tree **β₁: auxiliary tree** **β₂: auxiliary tree**

First step: Adjunction of α_1 in β_2:



γ₁: result of adjunction

Second step: Adjunction of β_1 in γ_1:



γ₂: result of adjunction, derivation complete.

Combinatory Categorical Grammars (CCGs)

CCG was developed by Mark Steedman in the early eighties. A CCG consists of a finite set of atomic categories, which can be combined by connectors / and \. By combining atomic categories with combinatory rules there will be generated complex categories. The lexicon of a CCG associates strings with categories. The connectors are used for some non-terminals A, B like: A / B or A \ B, where A / B means: To become an A, I need a B on the right, and A \ B means: To become an A, I need a B on the left.

Definition CCG

A CCG G is a quintuple $\{V_N, V_T, S, f, R\}$ where
V_N = finite set of non-terminals, the atomic categories
V_T = finite set of terminals, the lexical items
S = start symbol
f = function that maps terminals to non-terminals
R = combinatory rules

Definition combinatory rules

There a four types of combinatory rules, namely:

Application:

forward application:	$x/y\ y \Rightarrow x$
backward application:	$y\ x\backslash y \Rightarrow x$

Composition:

forward composition	$x/y\ y/z \Rightarrow x/z$
backward composition	$x\backslash y\ y\backslash z \Rightarrow x\backslash z$
forward crossing composition	$x/y\ y\backslash z \Rightarrow x\backslash z$
backward crossing composition	$x\backslash y\ y/z \Rightarrow x/z$

Coordination represents the string "and" like: $(x/x)\backslash x$

Type-raising:

forward type-raising	$x \Rightarrow y/(y\backslash x)$
backward type-raising	$x \Rightarrow y\backslash(y/x)$

Normal-form-CCGs only use the combinatory rules of application and composition.

Example:

Construct a CCG G for the sentence "ik haar hem de nijlpaarden zag helpen voeren":
G = $\{V_N, V_T, s, f, R\}$, where
V_N = {np; vp\np; vp\np/vp; s\np\np/vp}
V_T = {ik, haar, hem, de nijlpaarden, zag, helpen, voeren}
s = start symbol
f =
ik; haar; hem, the nijlpaarden := np
zag := s\np\np/vp
helpen := vp\np/vp
voeren := vp\np
R = {backward application ($*_1$), forward crossing composition ($*_2$)}

Derivation:

```
                                                                          helpen          voeren
                                                               zag        vp\np/vp          vp\np  *2
                                             de nijlpaarden   s\np\np/vp           vp\np\np  *2
                               hem              np                   s\np\np\np\np *1
                      har      np                             s\np\np\np  *1
   ik                 np                        s\np\np  *1
   np                          s\np  *1
                      s
```

In step one and two of the derivation, the np's were "collected" in the resulting categories, in the further steps the subjects to the np's were found and the np's were eliminated from the resulting categories. So finally only the start symbol s remains. That means the sentence could be derived correctly.

Linear Indexed Grammars (LIGs)

Linear indexed grammars were developed by Gazdar (1988) and are a special case of indexed grammars, which were developed by Aho in 1968.
LIGs are conform to context-free-grammars plus indices on the non-terminals, which describe the stack of the grammar. In contrast to IG, LIGs pass the indices only to one non-terminal. In the productions of the grammar there can be items pushed on the stack, and items can be removed from the stack.

Definition LIG
A LIG G is a quintuple $\{V_N, V_T, V_S, S, P\}$ where
V_N = finite set of non-terminals
V_T = finite set of terminals
V_S = finite set of stack symbols
S = start symbol
P = finite set of productions like $A\,[..x] \rightarrow \alpha_1 \ldots \alpha_n$

The following example shows how a LIG-derivation works:

Example
Grammar G = $\{V_N, V_T, V_S, S, P\}$ for sentence "ik haar hem de nijlpaarden zag helpen voeren"
V_N = {S, A, B, C}
V_T = {ik haar, hem, de nijlpaarden, zag, helpen, voeren, ε}
V_S = {h, v, z}
S = start symbol
P = {S[..]→ik haar A[..z];
A[..]→hem B[..h];
B[..]→de nijlpaarden C[..v];
C[..z]→C[..] zag,
C[..h]→ C[..]helpen,
C[..v]→ C[..]voeren,
C[ ]→ ε}

The rules 1-3 each push one item on the stack, the rules 4-6 each pop one item from stack, the last rule, maps empty stack to ε.

Derivation:
S[ ] → ik haar A [z]
→ ik haar hem B[zh]
→ ik haar hem de nijlpaarden C[zhv]
→ ik haar hem de nijlpaarden C[zh] voeren
→ ik haar hem de nijlpaarden C[z] helpen voeren
→ ik haar hem de nijlpaarden C[] zag helpen voeren
→ ik haar hem de nijlpaarden zag helpen voeren

Proof of equivalence of CCG, LIG and TAG

CCGs and TAGs generate the same class of string-languages and can also be described by LIGs. That is, the three grammar formalisms are weakly equivalent. The proof of equivalence will be arranged by circular inclusion. First I will show that for each CCG there is a LIG that describes the same language, then that for each LIG there is a TAG that describes the same language and last that for each TAG there is CCG that describes the same language. The proof is due to Vijay-Shanker and Weir (1994).

Step 1: CCG → LIG

The idea of the proof is, to interpret the CCG-categories as non-terminals plus stack, where the CCG-operation application corresponds to the stack operation push, the CCG-operation composition corresponds to the stack-operation pop (e.g. S → S[ ], S\a → S[\a]).
To construct a LIG for an existing CCG, every category of the CCG-lexicon and every CCG-rule must be transformed into a LIG-production. To this end, the CCG-rules must be turned around to become LIG-productions.

Example for the hippo-sentence
Given CCG G
G = {V_N, V_T, s, f, R}, where
V_N = {np; vp\np; vp\np/vp; s\np\np/vp}
V_T = {ik, haar, hem, de nijlpaarden, zag, helpen, voeren}
s = start symbol
f =
ik; haar; hem, the nijlpaarden := np
zag := s\np\np/vp
helpen := vp\np/vp
voeren := vp\np

For the derivation of the sentence “ik haar hem de nijlpaarden zag helpen voeren” only the combinatory-rules forward cross composition and backward application are used, therefore it is only necessary to show the transformation of these rules into LIG-productions.

LIG-productions for the CCG-lexicon:
np → np[]
vp → vp[\np]
vp → vp[\np/vp]
S → S[\np\np/vp]

LIG-productions for the CCG-rules:
Backward application (shown for zag:= s\np\np/vp)

CCG: np x\np → x	LIG: S[..]→ np S [..\np]

Forward cross composition (zag:=s\np\np/vp)

CCG: x/vp vp\(np/vp) → x\(np/vp)	LIG: S[..\np/vp] → S[../vp] vp[\np/vp]

Step 2: LIG → TAG

The idea for the proof starts from a normalized LIG, where every rule pushes or pops at most one item on the stack or from stack. Every stack is born empty and dies empty.

For the construction of the TAG the adjunction nodes are labeled with elementary stack operations:
- input non-terminal
- transition type
- output non-terminal

  where the transition type models the stack operation: no operation, push or pop.

For every combination of non-terminals, there must be constructed an initial tree of the form shown in **Fig. 6**, and for every rule of the form A [ ]→ x there must be constructed an initial tree of the form shown in **Fig.7,** where x' is obtained from x by removing the symbols representing the empty stack. Every derivation "begins" with an initial tree, such as shown in **Fig.6,** and "ends" with an initial tree such as shown in **Fig. 7.**.

```
      A
      |
[A, ε, B]/OA                 A
      |                      |
      B↓                     x'
```

Fig.6. initial tree for beginning a derivation

Fig.7. initial tree for ending a derivation

For every combination of non-terminals and symbol there must be constructed auxiliary trees to move between non-terminals, to do operations or to represent a LIG with no derivations.
The trees ensure that by moving between non-terminals the input-non-terminals are identical to the output-non-terminals and that for a stack operation an item, that is pushed on stack will also be removed from stack.
The auxiliary trees are shown in **Fig. 8.**, **Fig. 9.** and **Fig. 10.**.
Further auxiliary trees must be constructed to map the transition-types (no transition, push, pop).
These auxiliary trees were connected to the auxiliary tress from **Fig. 8**., **9.**, **10.**, and are shown in **Fig. 11-13**.

```
[A, ε, B]/NA
      |
[A, +a, C]/OA          [A, ε, B]/NA
      |                      |
[C, ε, D]/OA           [A, ε, C]/OA
      |                      |
[D, -a, B]/OA          [C, ε, B]/OA
      |                      |
[A, ε, B]/NA           [A, ε, B]/NA           [A, ε, A]/NA
```

Fig. 8. auxiliary tree to do stack operations

Fig.9. auxiliary tree to move between non-terminals

Fig.10. auxiliary tree for no derivation

```
LIG:      A[..] → x B[..] y         A[..]→ x B[..a] y          A[..a]→ x B[..] y

            [A, ε, B]/NA              [A, +a, B]/NA              [A, -a, B]/NA
TAG:       /     |      \            /     |       \            /     |      \
          x  [A, ε, B]/NA  y        x  [A, +a, B]/NA  y        x  [A, -a, B]/NA  y
```

Fig. 11. auxiliary tree for no stack operation

Fig. 12. auxiliary tree for push

Fig. 13. auxiliary tree for pop

Example for a construction of a TAG to a given LIG
Given a LIG for the language $L = \{a^n b^n c^n d^n\}$. For every LIG-rule it will be constructed a corresponding TAG-tree:

```
LIG-rule:                  ⇒            TAG-tree:

1) S[..] → a S[..i] d      ⇒            [S,+i,S]/NA
                                      /      |      \
                                     a  [S,+i,S]/NA  d


2) S[..]→ T[..]            ⇒            [S, ε, T]/NA
                                             |
                                        [S, ε, T]/NA


3) T[..i] → b T[..] c      ⇒            [S,+i,S]/NA
                                      /      |      \
                                     b  [S,+i,S]/NA  c


4) T[ ] → ε                ⇒                 T
                                             |
                                             ε
```

Next there will be constructed initial-trees for non-terminals S and T:

```
      S                  S                  T                  T
      |                  |                  |                  |
 [S, ε, S]/OA       [S, ε, T]/OA       [T, ε, S]/OA       [T, ε, T]/OA
      |                  |                  |                  |
      S↓                 T↓                 S↓                 T↓
```

The auxiliary trees that must be generated are not shown, because there are too many possibilities to construct the auxiliary trees. E. g. for two non-terminals and one stack symbol there will be created $2^4 = 16$ auxiliary trees.

Derivation of the word aabbccdd $\in$ L.
The derivation begins with the initial tree

```
      S
      |
 [S, ε, T]/OA
      |
      T↓
```

Now the auxiliary tree for stack operations will be adjoined into the initial tree, then the auxiliary tree for rule 1 pushes an item on the stack and in the next step the auxiliary tree for rule 3 pops an item from the stack to derive abcd:

```
         S
         |
   [S, ε, T]/NA
         |
   [S, +i, S]/OA
         |
   [S, ε, T]/OA
         |
   [T, -i, T]/OA
         |
   [S, ε, T]/NA
         |
         T↓
```

```
           S
           |
     [S, ε, T]/NA
           |
     [S, +i, S]/NA
   /       |       \
a   [S, +i, S]/NA   d
           |
     [S, ε, T]/OA
           |
     [T, -i, T]/OA
           |
     [S, ε, T]/NA
           |
           T↓
```

```
           S
           |
     [S, ε, T]/NA
           |
     [S, +i, S]/NA
   /       |       \
a   [S, +i, S]/NA   d
           |
     [S, ε, T]/OA
           |
     [T, -i, T]/NA
   /       |       \
b   [T, -i, T]/NA   c
           |
     [S, ε, T]/NA
           |
           T↓
```

Next three steps work analogue to the three steps before: we adjoin the auxiliary tree for stack operation, and then the auxiliary tree for rule 1 and then for rule 3, deriving aabbccdd.

```
           S
           |
     [S, ε, T]/NA
           |
     [S, +i, S]/NA
   /       |       \
a   [S, +i, S]/NA   d
           |
     [S, ε, T]/NA
           |
     [S, +i, S]/OA
           |
     [S, ε, T]/OA
           |
     [T, -i, T]/OA
           |
     [S, ε, T]/NA
           |
     [T, -i, T]/NA
   /       |       \
b   [T, -i, T]/NA   c
           |
     [S, ε, T]/NA
           |
           T↓
```

```
           S
           |
     [S, ε, T]/NA
           |
     [S, +i, S]/NA
   /       |       \
a   [S, +i, S]/NA   d
           |
     [S, ε, T]/NA
           |
     [S, +i, S]/NA
   /       |       \
a   [S, +i, S]/NA   d
           |
     [S, ε, T]/OA
           |
     [T, -i, T]/OA
           |
     [S, ε, T]/NA
           |
     [T, -i, T]/NA
   /       |       \
b   [T, -i, T]/NA   c
           |
     [S, ε, T]/NA
           |
           T↓
```

```
           S
           |
     [S, ε, T]/NA
           |
     [S, +i, S]/NA
   /       |       \
a   [S, +i, S]/NA   d
           |
     [S, ε, T]/NA
           |
     [S, +i, S]/NA
   /       |       \
a   [S, +i, S]/NA   d
           |
     [S, ε, T]/OA
           |
     [T, -i, T]/NA
   /       |       \
b   [T, -i, T]/NA   c
           |
     [S, ε, T]/NA
           |
     [T, -i, T]/NA
   /       |       \
b   [T, -i, T]/NA   c
           |
     [S, ε, T]/NA
           |
           T↓
```

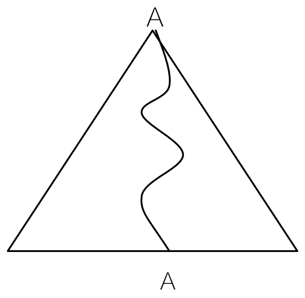
In the last two steps of the derivation, we adjoin the auxiliary tree for rule 2 and then the initial tree for rule 4.

```
          S                                   S
          |                                   |
    [S, ε, T]/NA                        [S, ε, T]/NA
          |                                   |
    [S, +i, S]/NA                       [S, +i, S]/NA
  /       |       \                   /       |       \
a   [S, +i, S]/NA   d               a   [S, +i, S]/NA   d
          |                                   |
    [S, ε, T]/NA                        [S, ε, T]/NA
          |                                   |
    [S, +i, S]/NA                       [S, +i, S]/NA
 /        |       \                   /       |       \
a   [S, +i, S]/NA   d               a   [S, +i, S]/NA   d
          |                                   |
    [S, ε, T]/NA                        [S, ε, T]/NA
          |                                   |
    [T, -i, T]/NA                       [T, -i, T]/NA
  /       |       \                   /       |       \
b   [T, -i, T]/NA   c               b   [T, -i, T]/NA   c
          |                                   |
    [S, ε, T]/NA                        [S, ε, T]/NA
          |                                   |
    [T, -i, T]/NA                       [T, -i, T]/NA
  /       |       \                   /       |       \
b   [T, -i, T]/NA   c               b   [T, -i, T]/NA   c
          |                                   |
    [S, ε, T]/NA                        [S, ε, T]/NA
          |                                   |
          T↓                                  T↓
                                              |
                                              ε
```

Step 3: Proof TAG → CCG

The basic idea here affects the correspondence between CCG- and TAG-operations.
The TAG-operation adjunction corresponds to the CCG-function composition, the TAG-operation substitution corresponds to the CCG-function application. To construct a CCG to a given TAG, the TAG must be available in normal-form (all TAGs can be transferred to normal form).
TAG in normal-form, means

– at most binary branching
– all internal nodes are either OA or NA
– all OA-nodes are on the spine or are sisters of nodes of the spine, where the spine of a tree is the path from root A of the tree to the foot A of the tree:



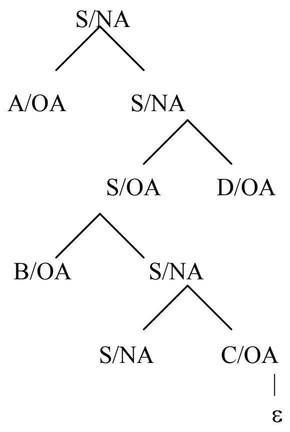
Normal form TAGs do not use substitution, but model substitution as we have described above by adjunction.

To construct the CCG every TAG-auxiliary tree must be transferred by an algorithm to a CCG-category and in addition a CCG- ^ -category. The CCG-^-categories must be built because the original TAG distinguishes lexical trees and non-lexical-trees. Lexical trees can only be adjoined into sisters of the spine and thus model substitution, non-lexical-trees can be adjoined in both the sisters of the spine or the spine itself. In CCG, TAG-adjunctions of non-lexical auxiliary trees into the spine of other non-lexical auxiliary trees are modeled by functional composition. The ^ - categories are used to forbid adjunction of lexical auxiliary trees into the spine of non-lexical trees.

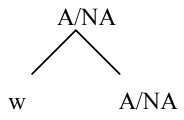
Algorithm for transferring TAG-auxiliary-trees to CCG-categories:
Initialising:
- pos = root (t) = A
- c = A or c = Â

until the foot of the tree is reached:
- if the non-spine daughter of pos is a left daughter with label B/OA, add /B to c
- if the non-spine daughter of pos is a right daughter with label B/OA, add \B to c
- if the spine daughter of pos has label C/OA, add /^C to c
- pos = spine daughter of pos

Example for the algorithm



c = S\A/D/^S\B/C
c = ^S\A/D/^S\B/C

Construction of a CCG from a TAG in normal-form:
Lexicon:
- for every lexical auxiliary tree of the form  construct an lexicon-item w := A

- for every from a non-lexical auxiliary tree constructed category c, construct an lexicon-item ε := c

rules:
- forward- and backward-application model adjunction of auxiliary trees into sisters of the spine (models substitution):
  $x/A \ A \rightarrow x$
  $A \ x\backslash A \rightarrow x$
- composition models the adjunction of non-lexical trees into the spine:
  for every non-terminal A and $i \leq n$, with n = length of the longest spine in auxiliary tree of the TAG:
  $x/\hat{A} \ \hat{A}|_1 z_1|_2 \ldots |_i z_i \rightarrow x|_1 z_1|_2 \ldots |_i z_i$ $\quad (| \in \{/,\backslash\}$

The constructed CCG describes the same language as the TAG, and so it is shown that TAG, CCG and LIG describe the same string-languages by circular inclusion.

Linear Context-Free Rewriting Systems (LCFRSs)

Linear context-free rewriting systems arose from the observation that a number of grammatical formalisms share two properties:

– Their derivation trees can be generated by a context-free grammar
– Their composition operations are size-preserving, i.e., when two or more substructures are combined only a bounded amount of structure is added or deleted

Definition LCFRS

A linear context-free rewriting system G is a Quintuple {V, F, P, R, S}, with
V = a finite set of non-terminals
F = finite set of function symbols
P = finite set of productions like $A \rightarrow f(A_1, \ldots, A_n)$, where $n \geq 0$, $f \in F$, $A_1, \ldots, A_n \in V$
S = start symbol
R a regular and total function $f_G(<w_{1,1},\ldots,w_{1,k1}>,\ldots,<w_{n,1},\ldots,w_{n,kn}>) = <t_1, \ldots, t_k>$, with $n \geq 0$, every t_i is a chain of $w_{m,n}$ and a fix amount of terminals

Example

Grammar G = {V, F, P, R, S} for sentence "ik haar hem de nijlpaarden zag helpen voeren"
V = { w_1, w_2}
F = {start, add_{ik}, add_{haar}, add_{hem}, $add_{de\ nijlpaarden}$}
R =
– start() = <ε, ε>
– add_{ik} (<w_1, w_2>) = <w_1 ik, w_2>
– add_{haar} (<w_1, w_2>) = <w_1 haar, w_2 zag>
– add_{hem} (<w_1, w_2>) = <w_1 hem, w_2 helpen>
– $add_{de\ nijlpaarden}$ (<w_1, w_2>) = <w_1 de nijlpaarden, w_2 voeren>
– concatenate (<w_1, w_2>) = < w_1w_2>
P =
– S → $start_\varepsilon$()
– S → add_{ik} (S)
– S → add_{haar} (S)
– S → add_{hem} (S)
– S → $add_{de\ nijlpaarden}$ (S)
– S → concatenate
S = start symbol

Derivation:
S → <ε, ε>
→ <ik, ε>
→ <ik haar, zag>
→ <ik haar hem, zag helpen>
→ <ik haar hem de nijlpaarden, zag helpen voeren>
→ <ik haar hem de nijlpaarden zag helpen voeren>

The given LCFRS derives the sentence.

Multiple Context-Free Grammars (MCFGs)

Multiple context-free grammars were introduced in the late 80's as a very expressive formalism, subsuming linear context-free rewriting systems and other mildly context-sensitive formalisms, but still with a polynomial parsing algorithm. MCFG is an instance of generalized CFG, with the following restrictions on linearizations:

- Linearization types are restricted to tuples of strings
- The only allowed operations in linearization functions are tuple projections and string concatenations

Definition MCFG

A MCFG G is a quintuple $\{V_N, V_T, P, F, S\}$ with
V_N = a finite set of non-terminals
V_T = a finite set of terminals
P = a finite set of rules
F = total, regular function, that derives strings to strings
S = start symbol

Example

Grammar G = $\{ V_N, V_T, P, F, S\}$ for the language $L = \{a^n b^n c^n d^n\}$
$V_N = \{S, A\}$
$V_T = \{a, b, c, d\}$
$P = \{S \rightarrow \text{conc}(A), A \rightarrow f(A) \mid g(\,)\}$
F = {conc, f, g}

conc: $<x_1, x_2> \rightarrow < x_1x_2>$
f: $<x_1, x_2> \rightarrow < ax_1b, cx_2d>$
g: $<x1,x2> \rightarrow <\varepsilon, \varepsilon>$

The given grammar describes the language L.

Conclusion

Mildly context-sensitive grammar-formalisms are a weak extension of the context-free grammars, with the goal to describe non-context-free phenomenons like cross-serial dependencies, but still being efficiently, i.e., polynomially parsable. In this paper are shown TAG, CCG, LIG, LCFRS and MCFG. The languages that are described by TAG, CCG or LIG are a proper subset of the languages that are described by LCFRS or MCFG. The grammar formalisms TAG, CCG, LIG and LCFRS respectively MCFG are based on absolutely different kind of paradigms. TAG is a tree-substitution-system, CCG is a categorial grammar, LIG is a CFG with stack, LCFRS and MCFG are generalized CFGs. The proof of Vijay-Shanker and Weir (1994) shows, that TAG, CCG and LIG are equivalent according to the string languages they generate.

References

Alfred V. Aho 1968: Indexed Grammars and Extension of Context-Free Grammars.
Journal of the Association for Computing Machinery 15, 647-671

Bar-Hillel, Yehoshua 1953: A Quasi-Arithmetical Notation for Syntactic Description Language. 29, 4758

Gerald Gazdar, 1988: Applicability of Indexed Grammars to Natural Languages
In U. Reyle and C. Rohrer, editors, natural language parsing and linguistic theories, pages 69-94

Klaus von Heusinger: Kategoriale Unifikationsgrammatik. 2005

Gerhard Jäger & Jens Michaelis (Sommerschule ESSLLI 2004 Nancy)
An Introduction to Mildly Context-Sensitive Grammar Formalisms.

Aravind K. Joshi, Leon S. Levy, Masako Takahashi 1975: Tree Adjunct Grammars.
Journal of computer and system science. 10(1):136-163

Aravind K. Joshi, K. Vijay-Shanker 1985: Some Computional Properties of Tree Adjoining Grammars.
ACL 1985: 82-93

Volker Kaatz: Linear Indizierte Grammatiken und Sprachen. 2005

Mark Steedman 2000: The Syntactic Process. Cambridge, MA, MIT Press.

K. Vijay-Shanker, David J. Weir 1993: Parsing Some Constrained Grammar Formalisms.
Computional Linguistics 19(4):591-636

K. Vijay-Shanker, David J. Weir 1994: The Equivalence of Four Extensions of Context-Free Grammars,
Mathematical Systems Theory 27(6):511-546