

Conversion between four extensions of context-free grammars

Matthijs Melissen

Contents

1	Introduction	1
2	Data types	2
2.1	General types	2
2.2	Linear indexed grammar	2
2.3	Combinatory categorial grammar	3
2.4	Head grammar	5
2.5	Tree adjoining grammar	6
3	Conversions	7
3.1	Ccg to lig	7
3.2	Lig to hg	10
3.3	Hg to tag	14
3.4	Tag to ccg	17
4	Example grammars	24
4.1	Linear indexed grammar	24
4.2	Combinatory categorial grammar	24
4.3	Head grammar	24
4.4	Tree adjoining grammar	25
5	Use of the program	25
6	Conclusion	25

Abstract

In this paper, an algorithm is presented which is capable of translating back and forth between four equivalent extensions of Context-Free Grammars. These grammars are Combinatory Categorical Grammars, Head grammars, Linear Indexed Grammars and Tree Adjoining Grammars. The algorithm does work, but turns out to be highly inefficient.

1 Introduction

Several extensions of context-free grammars, so called weakly context-sensitive grammars, have been described during the last decennia. Most important among them are combinatory categorial grammars (ccg) [Steedman, 1986], head grammars (hg) [Pollard, 1984], linear indexed grammars (lig) [Gazdar, 1988] and tree adjoining grammars (tag) [Levy and Takahashi, 1975]. In the paper "The equivalence of four extensions of context-free grammars" [Vijay-Shanker and Weir, 1994], the equivalence of those four formalisms is proven by giving a way of converting these formalisms into each other.

In this paper, I will investigate whether this procedure is suitable to actually convert these kinds of grammars into others. I will do so by presenting an Haskell program which does this job.

I assume the reader is already familiar with the four formalisms in question.

I will present the complete code in this paper. Every fragment is preceded by explanation about the next part of code.

2 Data types

First we define the module, called Equivalence, and import the predefined modules Char and List.

```
module Equivalence where

import Char
import List
```

We use two generic functions. The function `s1` is used to present a list without commas. The function `powerset` returns the powerset of its input.

```
s1 :: [String] -> String
s1 [] = ""
s1 (x:xs) = x ++ s1 xs

powerset :: [a] -> [[a]]
powerset [] = [[]]
powerset (x:xs) = xss ++ map (x:) xss
                where xss = powerset xs
```

2.1 General types

We start by defining general data types for nonterminals and terminals, of which all four grammars make use.

```
type NonTerm = String
type Term = String
```

2.2 Linear indexed grammar

We continue by defining the datatypes for linear indexed grammars. We represent a Linear Indexed Grammar as a five tuple $G = (V_N, V_T, V_I, S, P)$ where V_N is a finite set of nonterminals, V_T is a finite set of terminals, V_I is a finite set of indexes (stack symbols), $S \in V_N$ is the start symbol and P is a finite set of productions. The notation $A[\circ \circ \eta]$ stands for nonterminal A with the string of indices η on top of the stack.

```
data Lig = Lig [NonTerm] [Term] [Index] NonTerm [LigProd]
instance Show Lig where
  show (Lig vn vt vi s p) =
    "Nonterminals: " ++ show vn ++ "\n" ++
    "Terminals: " ++ show vt ++ "\n" ++
    "Indices: " ++ show vi ++ "\n" ++
    "Start Symbol: " ++ s ++ "\n" ++
    "Productions: \n" ++ showProductions p ++ "\n"

type Index = String
```

Productions can have two forms. The first form is the "nonterminal production"

$$A[\circ \circ \eta] \rightarrow \psi A'[\circ \circ \eta'] \psi'$$

where $A, A' \in V_N, \eta, \eta' \in V_I^*, \psi, \psi' \in (V_N[V_I^*] \cup V_T)^*$. We represent this in Haskell as follows:

$$\text{LPNT}(A, \eta)(\psi, A, \eta, \psi')$$

The second form is the "terminal production"

$$A[\eta] \rightarrow \psi$$

where $A \in V_N, \eta \in V_I^*, \psi \in (V_N[V_I^*] \cup V_T)^*$. This kind of production will be represented as

$$\text{LPT}(A, \eta)(\psi)$$

```
data LigProd = LPNT (NonTerm, [Index]) ([Symbol], NonTerm, [Index], [Symbol])
              | LPT (NonTerm, [Index]) [Symbol]
instance Show LigProd where
  show (LPNT (v1,i1) (s1,v2,i2,s2)) = v1 ++ "[oo" ++ (sl i1) ++ "]" --> " ++
    (showSymbolList s1) ++ v2 ++ "[oo" ++ (sl i2) ++ "]" ++ " " ++ (showSymbolList s2)
  show (LPT (v,i) (s)) = v ++ "[" ++ (sl i) ++ "]" --> " ++ (show s)
instance Eq LigProd where
  LPT (v, i) s == LPT (v', i') s'
    = v==v' && i==i' && s==s'
  LPNT (v1, i1) (s1, v2, i2, s2) == LPNT (v1', i1') (s1', v2', i2', s2')
    = v1==v1' && i1==i1' && s1==s1' && v2==v2' && i2==i2' && s2==s2'
  a == b
    = False

showProductions [] = ""
showProductions (x:xs) = show x ++ "\n" ++ showProductions xs
```

We still have to define the symbol data type, which we used as shorthand for $V_N[V_I^*] \cup V_T$.

```
data Symbol = SNT NonTerm [Index] | ST Term
instance Show Symbol where
  show (SNT v i) = v ++ "[" ++ sl i ++ "]"
  show (ST v) = v
instance Eq Symbol where
  SNT v i == SNT v' i' = v == v' && i == i'
  ST v == ST v' = v == v'
  x == y = False

showSymbolList [] = ""
showSymbolList (x:xs) = show x ++ " " ++ showSymbolList xs
```

2.3 Combinatory categorial grammar

Now we define the combinatory categorial grammars. A combinatory categorial grammar is a five tuple $G = (V_N, V_T, S, f, R)$. where V_N is a finite set of nonterminals, V_T is a finite set of terminals, S is a distinguished member of V_N , f is a function that maps elements of V_T^c to finite subsets of formulas and R is a finite subset of productions.

```

data Ccg = Ccg [NonTerm] [Term] NonTerm [CcgFunctionPair] [CcgRule]
instance Show Ccg where
  show (Ccg vn vt s f r) =
    "Nonterminals: " ++ show vn ++ "\n" ++
    "Terminals: " ++ show vt ++ "\n" ++
    "Start Symbol: " ++ s ++ "\n" ++
    "Function: " ++ show f ++ "\n" ++
    "Rules: \n" ++ showRules r ++ "\n"

showRules [] = ""
showRules (c:cs) = show c ++ "\n" ++ showRules cs

data CcgFunctionPair = CF Term [CcgForm]
instance Show CcgFunctionPair where
  show (CF x y) = "f(" ++ x ++ ")=" ++ show y

```

Productions can have one of these forms: $(x/y)(\dots(y|_1z_1)|_2\dots|_nz_n) \rightarrow (\dots(x|_1z_1)|_2\dots|_nz_n)$ or $(\dots(y|_1z_1)|_2\dots|_nz_n)(x/y) \rightarrow (\dots(x|_1z_1)|_2\dots|_nz_n)$ with $|_i \in /, \backslash$. The first one we represent by `CcgRule R x y [(|1, z1), (|2, z2), ..., (|n, zn)]`, the second one by `CcgRule L x y [(|1, z1), (|2, z2), ..., (|n, zn)]`. The variable x can be a normal variable, or a target restricted variable $\overset{A}{x}$ with $A \in V_N$, which we denote in the Haskell output by `xA`.

```

data CcgRule = CcgRule CcgOp CcgForm CcgForm [(CcgOp, CcgForm)]
instance Show CcgRule where
  show (CcgRule R x y zs) = "(" ++ showX x ++ "/" ++ show y ++ ")" (" ++ show y ++
    showFormulaList zs ++ ") --> (" ++ showX x ++ showFormulaList zs ++ ")"
  show (CcgRule L x y zs) = "(" ++ show y ++ showFormulaList zs ++ ")" (" ++ showX x ++
    "\\ " ++ show y ++ ") --> (" ++ showX x ++ showFormulaList zs ++ ")"
showX (A nt) = "x^" ++ nt
showX (V v) = v

```

In `cgg` we have two operators: the forward slash and the backward slash.

```

data CcgOp = R | L
instance Show CcgOp where
  show R = "/"
  show L = "\\ "
instance Eq CcgOp where
  R==R = True
  L==L = True
  x==y = False

type CcgVar = String

```

A formula in CCG can be either a nonterminal, a variable (which can be substituted for any non-terminal) or two formulas connected by an operator. Unlike in normal categorial grammars, in `cgg S\A` stands for a nonterminal S which misses a terminal A on the left, not a nonterminal A which lacks a nonterminal S .

```

data CcgForm = F CcgOp CcgForm CcgForm
  | A NonTerm
  | V CcgVar
instance Show CcgForm where
  show (F op x y) = show x ++ show op ++ show y

```

```

show (A x) = x
show (V x) = x
instance Eq CcgForm where
  F op x y == F op' x' y' = op==op' && x==x' && y == y'
  A x      == A x'      = x==x'
  V x      == V x'      = x==x'
  x        == y         = False

showFormulaList [] = ""
showFormulaList ((op,f):fs) = show op ++ show f ++ showFormulaList fs

```

2.4 Head grammar

A head grammar is a four tuple $G = (V_N, V_T, S, P)$ where V_N is a finite set of nonterminal symbols, V_T is a finite set of terminal symbols, $S \in V_N$ is the start symbol and P is a finite set of productions.

```

data Hg = Hg [NonTerm] [Term] NonTerm [HgProduction]
instance Show Hg where
  show (Hg vn vt s p) =
    "Nonterminals: " ++ show vn ++ "\n" ++
    "Terminals: " ++ show vt ++ "\n" ++
    "Start Symbol: " ++ s ++ "\n" ++
    "Productions: \n" ++ showHgProductions p ++ "\n"

```

A production in hg has the form $A \rightarrow f$, where f is a so called hg-function.

```

data HgProduction = HgProduction NonTerm HgFunction
instance Show HgProduction where
  show (HgProduction vn f) = vn ++ " --> " ++ show f
instance Eq HgProduction where
  HgProduction vn f == HgProduction vn' f' = vn == vn' && f == f'

showHgProductions [] = ""
showHgProductions (c:cs) = show c ++ "\n" ++ showHgProductions cs

```

Hg-functions can have two forms: the first form is $W(\sigma_1\sigma_2)$, and the second form $C_{i,j}(\sigma_1, \dots, \sigma_n)$ where σ_i is a hg-symbol. The integer j is equal to n , and i is the place in the formula where wrapping can occur.

```

data HgFunction = W (HgSymbol, HgSymbol)
                | C Int Int [HgSymbol]
instance Show HgFunction where
  show (W (s1, s2)) = "W (" ++ show s1 ++ ", " ++ show s2 ++ ")"
  show (C i j ss) = "C" ++ show i ++ "," ++ show j ++ " (" ++ show ss ++ ")"
instance Eq HgFunction where
  W (s1, s2) == W (s1', s2') = s1==s1' && s2==s2'
  C i j ss == C i' j' ss' = i==i' && j==j' && ss==ss'
  a == b = False

```

Hg-symbols have the form $V_N \cup (V_T^* \times V_T^*)$. A pair of two terminals is denoted $w_1 \uparrow w_2$, where \uparrow denotes the place the wrapping operation occurs.

```

data HgSymbol = HgNT NonTerm
              | HgT [Term] [Term]
instance Show HgSymbol where

```

```

show (HgNT nt) = nt
show (HgT t1 t2) = s1 t1 ++ "^" ++ s1 t2
instance Eq HgSymbol where
  HgNT nt == HgNT nt'      = nt==nt'
  HgT ws1 ws2 == HgT ws1' ws2' = ws1==ws1' && ws2==ws2'
  a == b                    = False

```

2.5 Tree adjoining grammar

Finally, a tree adjoining grammar is a five tuple $G = (V_N, V_T, S, \mathcal{I}, \mathcal{A})$, where V_N is a finite set of nonterminals, V_T is a finite set of terminals, $S \in V_N$ is a distinguished nonterminal, \mathcal{I} is a set of trees which we call the initial trees and \mathcal{A} is an ordered list of trees called the auxiliary trees. The trees in the auxiliary trees are labelled $1 \dots n$.

```

data Tag = Tag [NonTerm] [Term] NonTerm [Tree] [Tree]
instance Show Tag where
  show (Tag vn vt s init aux) =
    "Nonterminals: " ++ show vn ++ "\n" ++
    "Terminals: " ++ show vt ++ "\n" ++
    "Start Symbol: " ++ show s ++ "\n" ++
    "Initial trees: \n" ++ showTrees 1 init ++
    "Auxiliary trees: \n" ++ showTrees 1 aux

```

A tree is represented as a list of all the nodes in the tree.

```

data Tree = Tree [Node]
instance Show Tree where
  show (Tree []) = ""
  show (Tree ((loc,lab):ts)) = show loc ++ ": " ++ show lab ++ "; " ++ show (Tree ts)
showTrees :: Int -> [Tree] -> String
showTrees i [] = ""
showTrees i (t:ts) = show i ++ " = " ++ show t ++ "\n" ++ showTrees (i+1) ts

```

A node is a pair which contains both a location and a label. A location is represented as a list of integers, which gives us the number of the branch we have to look at. For example, $(2, 1)$ denotes the first child of the second child of the root node. Notice that the location of the root of a tree is the empty list.

```

type Node = (NodeLocation, NodeLabel)

type NodeLocation = [Int]

```

A label can either be empty (**EmptyNode**), a terminal (**TNode**) or a three tuple of a nonterminal, a specific adjoining (sa) property and an obligatory (oa) property (**NTNode**). Recall that all auxiliary trees are labelled with a natural number. The sa property is a list of those auxiliary trees which can be adjuncted at that node. If this list consists of all auxiliary trees, we denote this as AA. If a node does not have the sa property, it is said to be NA (non adjunctable). The oa property is true when adjoining is obligatory, and false otherwise.

```

data NodeLabel = NTNode NonTerm SA Bool
               | TNode Term
               | EmptyNode
instance Show NodeLabel where
  show (NTNode nt sa oa) = nt ++ "(" ++ show sa ++ showOa oa ++ ")"

```

```

where
  showOa True = " OA"
  showOa False = ""
  show (TNode t) = t
  show EmptyNode = "empty"
instance Eq NodeLabel where
  NNode nt na oa == NNode nt' na' oa' = nt == nt' && na == na' && oa == oa'
  TNode t == TNode t' = t == t'
  EmptyNode == EmptyNode = True
  x == t = False

data SA = SA [Int]
  | AA
instance Show SA where
  show (SA []) = "NA"
  show (SA xs) = show xs
  show (AA ) = ""
instance Eq SA where
  SA xs == SA xs' = xs == xs'
  AA == AA = True
  x == y = False

```

3 Conversions

Now I will present the actual conversions. I will give conversions from ccg to lig, from lig to hg, from hg to tag and from tag to ccg. By using just these four conversions, it is possible to convert any of the four grammars into one of the others.

3.1 Ccg to lig

We convert a combinatory categorial grammar into a linear indexed grammar by first converting it into a more simpler ccg, and then translating the simplified lig into a ccg.

```

ccg2lig :: Ccg -> Lig
ccg2lig x = ccg2lig' (simplifyCcg x)

```

To simplify the lig, we have to simplify all the rules.

```

simplifyCcg :: Ccg -> Ccg
simplifyCcg ccg@(Ccg vn vt s f r) = (Ccg vn vt s f r') where
  r' = concat (map (simplifyRule vn ar) r)
  ar = args ccg

```

Recall a rule has two parts, $(x|_0y)$ and $(\dots(y|_1z_1)|_2\dots|_nz_n)$, where $|_n$ is either a slash or a backslash. To simplify a rule, we have to do two things. The first thing is to change the symbol x . This is done by `simplifyX`. The second thing we do is replacing all variables in the rest of the rule for all possible 'arguments' from the ccg.

```

simplifyRule :: [NonTerm] -> [CcgForm] -> CcgRule -> [CcgRule]
simplifyRule vns args r =
  [ccgReplaceVar r'' var form | r'' <- r', var <- (ccgGetVars r''), form <- args] where
    r' = simplifyX r vns

```

An argument of a `ccg` is defined as the arguments of the resulting formulas of the `ccg`-function. The argument of a simple formula is defined as the empty list, and the argument of a complex formula as its right element. For example, $args(((A/B)\(C/D))\E) = [E, C/D, B]$.

```
args :: Ccg -> [CcgForm]
args (Ccg _ _ _ f _) = nub [c | (CF _ cs') <- f, c' <- cs', c <- args' c']

args' :: CcgForm -> [CcgForm]
args' (A a) = []
args' (F _ l r) = args' l ++ [r]
```

If the target symbol x (the first symbol) of the part x/y or $x\backslash y$ of the production is a variable, we add new productions in which we replace this x by variables restricted to all possible nonterminals.

```
simplifyX :: CcgRule -> [NonTerm] -> [CcgRule]
simplifyX r@(CcgRule dir (A a) y zs) vns = [r]
simplifyX (CcgRule dir (V v) y zs) vns = [CcgRule dir (A vn) y zs | vn <- vns]
```

The function `ccgGetVars r` makes a list of all variables occurring in the rule r by recursively walking through the data structure.

```
ccgGetVars :: CcgRule -> [CcgVar]
ccgGetVars (CcgRule _ _ (V h) xs) = nub ([h] ++ ccgGetVars' xs)
ccgGetVars (CcgRule _ _ f@(F _ _ _) xs) = nub (ccgGetVars'' f ++ ccgGetVars' xs)
ccgGetVars (CcgRule _ _ (A _) _) = []

ccgGetVars' :: [(CcgOp, CcgForm)] -> [CcgVar]
ccgGetVars' ((_, V v):rest) = [v] ++ ccgGetVars' rest
ccgGetVars' ((_, f@(F op x y)):rest) = ccgGetVars'' f ++ ccgGetVars' rest
ccgGetVars' ((_, A f):rest) = ccgGetVars' rest
ccgGetVars' [] = []

ccgGetVars'' :: CcgForm -> [CcgVar]
ccgGetVars'' (F _ l r) = ccgGetVars'' l ++ ccgGetVars'' r
ccgGetVars'' (A _) = []
ccgGetVars'' (V v) = [v]
```

The function `ccgReplaceVar r v f` recursively walks through the datastructure r , replacing all variables v for the formula f .

```
ccgReplaceVar :: CcgRule -> CcgVar -> CcgForm -> CcgRule
ccgReplaceVar (CcgRule dir x (V v') xs) v form | v == v'      = (CcgRule dir x form xs')
                                                    | otherwise    = (CcgRule dir x (V v') xs')
                where xs' = ccgReplaceVar' xs v form
ccgReplaceVar (CcgRule dir x f@(F _ _ _) xs) v form = (CcgRule dir x f' xs') where
    xs' = ccgReplaceVar' xs v form
    f' = ccgReplaceVar'' f v form

ccgReplaceVar' :: [(CcgOp, CcgForm)] -> CcgVar -> CcgForm -> [(CcgOp, CcgForm)]
ccgReplaceVar' ((op, V v'):r) v form | v == v' = ((op, form): (ccgReplaceVar' r v form))
                                        | v /= v' = ((op, V v'): (ccgReplaceVar' r v form))
ccgReplaceVar' ((op, f@(F o x y)):rest) v form = ((op, f'):rest') where
    f' = ccgReplaceVar'' f v form
    rest' = ccgReplaceVar' rest v form
ccgReplaceVar' [] _ _ = []
```

```

ccgReplaceVar'' :: CcgForm -> CcgVar -> CcgForm -> CcgForm
ccgReplaceVar'' (F op l r) v form = (F op l' r') where
    l' = ccgReplaceVar'' l v form
    r' = ccgReplaceVar'' r v form
ccgReplaceVar'' (A a) v form = A a
ccgReplaceVar'' (V v') v form | v'==v    = form
                               | otherwise = V v'

```

Now we are ready to actually convert the ccg. We do this as follows. Note that both the productions from the ccg as the functions from the ccg deliver us new productions.

```

ccg2lig' :: Ccg -> Lig
ccg2lig' (Ccg vn vt s f r) = (Lig vn' vt' vi' s' p') where
    vn' = vn
    vt' = vt
    vi' = ["/", "\\ ", "( ", ")"] ++ vn
    s' = s
    p' = map convertRule r ++ concat (map convertFunction f)

```

First we look at the case of the new lig productions produced by old tag productions. The function `enc` converts a ccg formula into a lig symbol. We will see how this is done later.

Example: $(x/T)(T\A/T\B) \rightarrow (x\A/T\B)$ is converted by this procedure to $S[\A/T\B] \rightarrow S[\circ \circ /T]T[\A/T\B]$.

```

convertRule :: CcgRule -> LigProd
convertRule (CcgRule R (A a) c0@(F _ _ _) eta) =
    LPNT (a, eta') ([], a, ["/"] ++ c0', [b]) where
        b = enc (headAndListToFormula c0 eta)
        c0' = formToIndices c0
        eta' = listToIndices eta
convertRule (CcgRule R (A a) (A c0) eta) =
    LPNT (a, eta') ([], a, ["/"] ++ c0', [b]) where
        b = enc (headAndListToFormula (A c0) eta)
        c0' = formToIndices (A c0)
        eta' = listToIndices eta
convertRule (CcgRule L (A a) c0@(F _ _ _) eta) =
    LPNT (a, eta') ([b], a, ["\\ "] ++ c0', []) where
        b = enc (headAndListToFormula c0 eta)
        c0' = formToIndices c0
        eta' = listToIndices eta
convertRule (CcgRule L (A a) (A c0) eta) =
    LPNT (a, eta') ([b], a, ["\\ "] ++ c0', []) where
        b = enc (headAndListToFormula (A c0) eta)
        c0' = formToIndices (A c0)
        eta' = listToIndices eta

convertFunction :: CcgFunctionPair -> [LigProd]
convertFunction (CF "" cs) = map makeProduction cs where
    makeProduction c = LPT (a, eta) [] where
        SNT a eta = enc c
convertFunction (CF w cs) = map makeProduction cs where
    makeProduction c = LPT (a, eta) [(ST w)] where
        SNT a eta = enc c

```

The function `enc` translates a `ccg` formula into a lig symbol. This is done by using the target variable as nonterminal of the lig-rule. For example, $\text{enc}(T \setminus A / T \setminus B) = T[\setminus A / T \setminus B]$.

```
enc :: CcgForm -> Symbol
enc (A x) = SNT x []
enc (F R c1 c2) = SNT a (eta ++ ["/"] ++ c2') where
  (SNT a eta) = enc c1
  c2' = formToIndices c2
enc (F L c1 c2) = SNT a (eta ++ ["\\"] ++ c2') where
  (SNT a eta) = enc c1
  c2' = formToIndices c2
```

Finally we have three functions which converts `CcgForms`, `CcgOperators` and lists of `(CcgOp, CcgForm)` to Strings, so they can be used as index.

```
formToIndices :: CcgForm -> [Index]
formToIndices (A x) = [x]
formToIndices (F op x y) = ["(" ++ formToIndices x ++ ")"] ++ [operatorToIndex op] ++ ["(" ++
  formToIndices y ++ ")"]

operatorToIndex :: CcgOp -> Index
operatorToIndex R = "/"
operatorToIndex L = "\\"

headAndListToFormula :: CcgForm -> [(CcgOp, CcgForm)] -> CcgForm
headAndListToFormula head ((op,newHead):rest) = F op subFormula newHead
  where subFormula = headAndListToFormula head rest
headAndListToFormula head [] = head

listToIndices :: [(CcgOp, CcgForm)] -> [Index]
listToIndices ((op,f):rest) = [operatorToIndex op] ++ (formToIndices f) ++
  listToIndices rest
listToIndices [] = []
```

3.2 Lig to hg

Now we will present a function which converts linear indexed grammars into head grammars. We do this by first converting the lig into an easier form, and then translate it into an hg.

```
lig2hg :: Lig -> Hg
lig2hg x = lig2hg' (simplifyLig x)
```

To simplify the Lig, we will simplify each of its productions. The new V_N consists of all nonterminals which occur in the rules. Simplifying a production is a four stage process.

```
simplifyLig :: Lig -> Lig
simplifyLig (Lig vn vt vi s p) = (Lig vn' vt vi s p') where
  p' = simplifyLigProductions p
  vn' = allLigNonterminals p'

simplifyLigProductions :: [LigProd] -> [LigProd]
simplifyLigProductions ps = nub ps''''
  where
    ps' = concat (map ligRemoveTerminalsAndIndexes ps)
    ps'' = concat (map oneIndexAtATime ps')
```

```

ps''' = concat (map removeCopyRule ps'')
ps'''' = concat (map removeEmptyProductions ps''')

```

In the first step, we eliminate terminals and all except one of the nonterminals with indices. For example, we translate $S[\circ\circ] \rightarrow aS[\circ\circ l]B[mn]$ into $S[\circ\circ] \rightarrow CS[\circ\circ l]D$, $C \rightarrow a$, $D[\circ\circ] \rightarrow B[\circ\circ mn]$.

Recall our productions were of one of the following forms: $A[\circ\circ\eta] \rightarrow \psi A'[\circ\circ\eta']\psi'$ and $A[\eta] \rightarrow \psi$ with $A, A' \in V_N, \eta, \eta' \in V_I^*, \psi, \psi' \in (V_N[V_I^*] \cup V_T)^*$. After this translation, our productions are either of one of these forms: $A[\circ\circ\eta] \rightarrow B^* A'[\circ\circ\eta'] B'^*$ and $A[\eta] \rightarrow B^*$ with $A, A' \in V_N, \eta, \eta' \in V_I^*, B, B' \in (V_N[\])^*$, either of the form $A[\eta] \rightarrow w^*$ with $w \in V_T^c$.

The function `ligRemoveTerminalsAndIndexes` iterates over all η .

The function `ligRemoveTerminalsAndIndexes'` iterates over all symbols and returns a new symbol which substitutes the old symbol and a new additional production.

```

ligRemoveTerminalsAndIndexes :: LigProd -> [LigProd]
ligRemoveTerminalsAndIndexes (LPNT h (s1, nt, i, s2)) =
  [LPNT h (s'1, nt, i, s'2)] ++ p1 ++ p2 where
    (s'1, p1) = ligRemoveTerminalsAndIndexes' s1
    (s'2, p2) = ligRemoveTerminalsAndIndexes' s2
ligRemoveTerminalsAndIndexes (LPT h s) = [LPT h s'] ++ p where
  (s', p) = ligRemoveTerminalsAndIndexes' s

ligRemoveTerminalsAndIndexes' :: [Symbol] -> ([Symbol], [LigProd])
ligRemoveTerminalsAndIndexes' (ST t : xs) = (s:ss, p:ps) where
  (ss, ps) = ligRemoveTerminalsAndIndexes' xs
  s = SNT ("_T" ++ t) []
  p = LPT (("T" ++ t), []) ([ST t])
ligRemoveTerminalsAndIndexes' (s@(SNT _ []) : xs) = (s:ss, ps) where
  (ss, ps) = ligRemoveTerminalsAndIndexes' xs
ligRemoveTerminalsAndIndexes' (s@(SNT nt is) : xs) = (s:ss, p:ps) where
  (ss, ps) = ligRemoveTerminalsAndIndexes' xs
  s = SNT ("_I" ++ nt ++ "_" ++ concat is) []
  p = LPNT ("I" ++ nt ++ "_" ++ concat is), [] ([], nt, is, [])
ligRemoveTerminalsAndIndexes' [] = ([], [])

```

In the second stage, we rewrite the productions in such a way that every production pushes or pops only one index at a time. For example, we rewrite $A[\circ\circ] \rightarrow B[\circ\circ kl]$ into $A[\circ\circ] \rightarrow C[\circ\circ k]$ and $C[\circ\circ] \rightarrow B[\circ\circ l]$.

```

oneIndexAtATime :: LigProd -> [LigProd]
oneIndexAtATime (LPNT (nt1, []) (s1, nt2, [], s2)) =
  [LPNT (nt1, []) (s1, nt2, [], s2)]
oneIndexAtATime (LPNT (nt1, []) (s1, nt2, [i2], s2)) =
  [LPNT (nt1, []) (s1, nt2, [i2], s2)]
oneIndexAtATime (LPNT (nt1, []) (s1, nt2, (i2:i2s), s2)) =
  [LPNT (nt1, []) ([], nt', [i2], [])]
  ++ oneIndexAtATime (LPNT (nt', []) (s1, nt2, i2s, s2))
  where nt' = "_0a" ++ show s1 ++ "_" ++ nt2 ++ "_" ++ show i2s ++ show s2
oneIndexAtATime (LPNT (nt1, [i1]) (s1, nt2, [], s2)) =
  [LPNT (nt1, [i1]) (s1, nt2, [], s2)]
oneIndexAtATime (LPNT (nt1, i1s) (s1, nt2, i2s, s2)) =
  [LPNT (nt1, [i1]) ([], nt', [], [])]
  ++ oneIndexAtATime (LPNT (nt', i1r) (s1, nt2, i2s, s2))
  where
    nt' = "_0b" ++ show i1r ++ "_" ++ show s1 ++ "_" ++ nt2 ++ "_" ++ show i2s ++ show s2

```

```

    i1 = last i1s
    i1r = init i1s
oneIndexAtATime (LPT (a, []) [w]) =
  [LPT (a, []) [w]]
oneIndexAtATime (LPT (nt1, [i1]) ((SNT b []):bs)) =
  [LPNT (nt1, [i1]) ([],b,[],bs)]
oneIndexAtATime (LPT (nt1, []) []) =
  [LPT (nt1, []) []]
oneIndexAtATime (LPT (nt1, i1s) (bs)) =
  [LPNT (nt1, [i1]) ([],nt',[],[])]
  ++ oneIndexAtATime (LPT (nt', i1r) bs)
where
  nt' = "_0c" ++ show i1r ++ "_" ++ show bs
  i1 = last i1s
  i1r = init i1s

```

In the third stage, we eliminate copy rules: rules of the form $A[\circ] \rightarrow \psi[\circ]\psi'$. The variable nt' is again a fresh variable.

```

removeCopyRule :: LigProd -> [LigProd]
removeCopyRule (LPNT (nt1, []) (s1, nt2, [], s2)) =
  [LPNT (nt1, []) ([], nt', ["t"], []), LPNT (nt', ["t"]) (s1, nt2, [], s2)]
where
  nt' = "_Ca" ++ show s1 ++ "_" ++ nt2 ++ "_" ++ show s2
removeCopyRule (LPNT a b) = [LPNT a b]
removeCopyRule (LPT (nt, []) ((SNT b []):bs)) =
  [LPNT (nt, []) ([], nt', ["t"], []), LPNT (nt', ["t"]) ([], b, [], bs)]
where
  nt' = "_Cb" ++ b ++ "_" ++ show bs
removeCopyRule (LPT (nt, []) []) =
  [LPT (nt, []) []]
removeCopyRule (LPT (a, []) [ST w]) = [LPT (a, []) [ST w]]

```

Finally we have to remove empty indexes from productions which produce only the empty list:

```

removeEmptyProductions :: LigProd -> [LigProd]
removeEmptyProductions f@(LPT (nt, [i]) []) =
  [LPNT (nt, [i]) ([], nt', [], []), LPT (nt', []) []]
  where nt' = "_" ++ "E" ++ nt
removeEmptyProductions f@(LPT (_, _) _) = [f]
removeEmptyProductions f@(LPNT _ _) = [f]

```

The function `allLigNonterminals` gives a list of all nonterminals, by recursively walking through the productions.

```

allLigNonterminals :: [LigProd] -> [NonTerm]
allLigNonterminals ps = nub (concat (map allLigNonterminals' ps))

allLigNonterminals' :: LigProd -> [NonTerm]
allLigNonterminals' (LPNT (nt1, _) (ss1, nt2, _, ss2)) = [nt1] ++ [nt2] ++
  allLigNonterminals'' ss1 ++ allLigNonterminals'' ss2
allLigNonterminals' (LPT (nt, _) ss) = [nt] ++ allLigNonterminals'' ss

allLigNonterminals'' :: [Symbol] -> [NonTerm]
allLigNonterminals'' [] = []

```

```

allLigNonterminals'' (s:ss) = allLigNonterminals'' s ++ allLigNonterminals'' ss

allLigNonterminals''' :: Symbol -> [NonTerm]
allLigNonterminals''' (ST _) = []
allLigNonterminals''' (SNT nt _) = [nt]

```

After these four stages, we can guarantee that all productions have one of these forms, where $A, A_1, \dots, A_n \in V_N$, $l \in V_I$ and $w \in V_T^f$.

- $A[\circ] \rightarrow A_1[] \dots A_{i-1}[] Ai[\circ \circ l] A_{i+1}[] \dots A_n[]$
- $A[\circ \circ l] \rightarrow A_1[] \dots A_{i-1}[] Ai[\circ] A_{i+1}[] \dots A_n[]$
- $A[] \rightarrow w$

Now it is time to start the real conversion. This is done as follows. Every production from V_N is converted by `convertLigProduction`. We also add three kinds of 'terminal productions' `ps1`, `ps2` and `ps3`.

```

lig2hg' :: Lig -> Hg
lig2hg' (Lig vn vt vi s p) = (Hg vn' vt' s' p') where
  vn' = vn ++ [(show [a1, a2, eta]) | a1 <- vn, a2 <- vn, eta <- (vi ++ ["e"])]
  vt' = vt
  s' = s
  p' = concat (map (convertLigProduction vn) p) ++ ps1 ++ ps2 ++ ps3 where
    ps1 = [HgProduction (show [a, b, eta])
            (W (HgNT (show [a, c, "e"]), HgNT (show [c, b, eta])))
            | eta <- (vi ++ []), a <- vn, b <- vn, c <- vn]
    ps2 = [HgProduction (show [a, a, ""]) (C 1 1 [HgT [] []]) | a <- vn]
    ps3 = [HgProduction a (W (HgNT (show [a, b, ""]), HgNT b)) | a <- vn, b <- vn]

```

Now we are going to convert each of the productions of `Lig` into a production of `Hg`. The different instances of this function correspond to the different types in the list above.

```

convertLigProduction :: [NonTerm] -> LigProd -> [HgProduction]
convertLigProduction vn (LPNT (a, []) (aas, ai, [l], azs)) =
  [HgProduction (show [a, b, "e"]) (C i n (ss b)) | b <- vn] where
    i = (length aas) + 1
    n = (length aas) + 1 + (length azs)
    ss b = (map ligSymbolToHgSymbol aas) ++ [HgNT (show [ai, b, l])] ++
            (map ligSymbolToHgSymbol azs)
convertLigProduction vn (LPNT (a, [l]) (aas, ai, [], azs)) =
  [HgProduction (show [a, b, l]) (C i n (ss b)) | b <- vn] where
    i = (length aas) + 1
    n = (length aas) + 1 + (length azs)
    ss b = (map ligSymbolToHgSymbol aas) ++ [HgNT (show [ai, b, "e"])] ++
            (map ligSymbolToHgSymbol azs)
convertLigProduction vn (LPT (a, []) [ST w]) = [HgProduction a (C 1 1 [HgT [] [w]])]
convertLigProduction vn (LPT (a, []) []) = [HgProduction a (C 1 1 [HgT [] []])]

```

Finally we have a basic function which casts `lig` symbols to `hg` symbols.

```

ligSymbolToHgSymbol :: Symbol -> HgSymbol
ligSymbolToHgSymbol (SNT nonterm []) = HgNT nonterm

```

3.3 Hg to tag

The third conversion is a conversion from hg to tag. Again we first bring the hg into a form which is more easy to handle, and then convert this simplified hg to a tag.

```
hg2tag :: Hg -> Tag
hg2tag x = hg2tag' (simplifyHg x)
```

The process of simplifying the hg consists of three steps.

```
simplifyHg :: Hg -> Hg
simplifyHg (Hg vn vt s p) = (Hg vn' vt s (nub p''')) where
  p' = concat (map makeHgRulesBinary p)
  p'' = concat (map removeHgTerminals p')
  p''' = concat (map oneHgTerminalAtATime p'')
  vn' = allHgNonterminals p'''
```

First we make sure all productions are binary branching. Recall that binary rules have the form $A \rightarrow W(\sigma_1\sigma_2)$ or $A \rightarrow C_{i,j}(\sigma_1, \dots, \sigma_n)$ where $\sigma_n \in V_N \cup (V_T^* \times V_T^*)$. W -rules are already binary branching, so we just have to convert the C -rules. Informally, the following function recursively branches one σ off at a time.

For example, the rule $S \rightarrow C_{2,3}(A, B, C)$ is converted into $S \rightarrow C_{2,2}(A, D)$ and $D \rightarrow C_{1,2}(B, C)$.

The first instance of this function says we do not need to change the wrapping production. The second instance extends C -productions producing just a single symbol with the empty symbol. The third instance accepts productions which are already binary branching. The fourth instance splits off the first symbol where wrapping can be applied, and results in a production which says wrapping can be applied in the split off symbol. The fifth symbol does the same, but for the case wrapping can't be applied to the symbol which is split off. In the last two productions, nt' is a fresh variable.

Now all productions have the form $A \rightarrow W\sigma_1\sigma_2$ or $A \rightarrow C_{i,j}\sigma_1, \sigma_2$ with $\sigma_n \in V_N \cup (V_T^* \times V_T^*)$.

```
makeHgRulesBinary :: HgProduction -> [HgProduction]
makeHgRulesBinary p@(HgProduction _ (W _)) = [p]
makeHgRulesBinary p@(HgProduction nt (C 1 1 [s])) =
  [HgProduction nt (C 1 2 [s, HgT [] []])]
makeHgRulesBinary p@(HgProduction _ (C _ 2 _)) = [p]
makeHgRulesBinary (HgProduction nt (C 1 j (s:ss))) =
  [HgProduction nt (C 1 2 [s, HgNT nt'])] ++
  makeHgRulesBinary (HgProduction nt' (C 1 (j-1) ss))
  where nt' = "_B" ++ show j ++ "_" ++ show ss
makeHgRulesBinary (HgProduction nt (C i j (s:ss))) =
  [HgProduction nt (C 2 2 [s, HgNT nt'])] ++
  makeHgRulesBinary (HgProduction nt' (C 1 (j-1) ss))
  where nt' = "_B" ++ show j ++ "_" ++ show ss
```

Now we eliminate all terminals from these productions, and put them into new productions of the form $A \rightarrow C_{1,1}(ws_1 \uparrow ws_2)$ with $A \in V_N, ws_1, ws_2 \in V_T^*$. The function `removeHgTerminals` applies `removeHgTerminals'` to both symbols in the production. This latter function returns for each symbol a new (nonterminal) symbol, and possibly a new production, which converts the nonterminal symbol into the terminal.

For example, $A \rightarrow B(S, v \uparrow w)$ is converted into $A \rightarrow B(S, A')$ and $A' \rightarrow C_{1,1}(v \uparrow w)$.

Now all productions have either the form of the new productions, or the form $A \rightarrow W(\sigma_1\sigma_2)$ or $A \rightarrow C_{i,j}(\sigma_1, \sigma_2)$ with $A \in V_N, \sigma_n \in V_N$.

```
removeHgTerminals :: HgProduction -> [HgProduction]
removeHgTerminals (HgProduction a (C i 2 [b, c])) =
```

```

[HgProduction a (C i 2 [bs, cs])] ++ bp ++ cp where
  (bs,bp) = removeHgTerminals' b
  (cs,cp) = removeHgTerminals' c
removeHgTerminals (HgProduction a (W (b, c))) =
  [HgProduction a (W (bs, cs))] ++ bp ++ cp where
    (bs,bp) = removeHgTerminals' b
    (cs,cp) = removeHgTerminals' c

removeHgTerminals' :: HgSymbol -> (HgSymbol, [HgProduction])
removeHgTerminals' (HgNT nt) = (HgNT nt, [])
removeHgTerminals' t@(HgT ws1 ws2) = (HgNT nt', [HgProduction nt' (C 1 1 [t])])
  where nt' = "_T" ++ concat ws1 ++ "_" ++ concat ws2

```

Finally, we want to make sure productions to terminals have either 0 or 1 terminals on both sides of the upward arrow. We do this by recursively putting an outer terminal into an own production and create a new production which produces both this own production and the old production.

For example $A \rightarrow C_{1,1}(uv \uparrow wx)$ is first translated into $A \rightarrow C_{2,2}(B, C)$, $B \rightarrow (u \uparrow \epsilon)$ and $C \rightarrow (v \uparrow wx)$. Next, this last translation is rewritten into $C \rightarrow C_{1,2}(B, C)$, $C \rightarrow (x \uparrow \epsilon)$ and $B \rightarrow (v \uparrow w)$.

All our productions now have one of these forms, with $A, \sigma_n \in V_N, w_1, w_2 \in V_T$:

- $A \rightarrow C_{1,1}(w_1 \uparrow w_2)$
- $A \rightarrow W\sigma_1\sigma_2$
- $A \rightarrow C_{i,j}\sigma_1, \sigma_2$

```

oneHgTerminalAtATime :: HgProduction -> [HgProduction]
oneHgTerminalAtATime f@(HgProduction nt (C 1 1 [HgT ws1 ws2]))
  | length ws1 > 1 = HgProduction nt (C 2 2 [HgNT nt', HgNT nt''])
    : HgProduction nt' (C 1 1 [HgT [head ws1] []])
    : oneHgTerminalAtATime (HgProduction nt'' (C 1 1 [HgT (tail ws1) ws2]))
  | length ws2 > 1 = HgProduction nt (C 1 2 [HgNT nt''', HgNT nt'''])
    : HgProduction nt''' (C 1 1 [HgT [last ws2] []])
    : oneHgTerminalAtATime (HgProduction nt'''' (C 1 1 [HgT ws1 (init ws2)]))
  | otherwise      = [f]
  where
    nt' = "_0a" ++ head ws1
    nt'' = "_0b" ++ show (tail ws1) ++ "_" ++ show ws2
    nt''' = "_0c" ++ last ws2
    nt'''' = "_0d" ++ show ws1 ++ "_" ++ show (init ws2)
oneHgTerminalAtATime f = [f]

```

The function `allHgNonterminals` gives a list of all nonterminals, by iterating over the datastructure.

```

allHgNonterminals :: [HgProduction] -> [NonTerm]
allHgNonterminals ps = nub (concat (map allHgNonterminals' ps))

allHgNonterminals' :: HgProduction -> [NonTerm]
allHgNonterminals' (HgProduction nt (W (s1, s2))) =
  nt : (allHgNonterminals'' s1 ++ allHgNonterminals'' s2)
allHgNonterminals' (HgProduction nt (C _ _ ss)) = nt : concat (map allHgNonterminals'' ss)

allHgNonterminals'' :: HgSymbol -> [NonTerm]
allHgNonterminals'' (HgNT nt) = [nt]
allHgNonterminals'' (HgT _ _) = []

```

Now we can convert the simplified hg into a tag. We do this as follows. Notice that the initial tree is a tree with s as root and the empty node as child. For every production, we generate an auxiliary tree by `generateAux`.

```
hg2tag' :: Hg -> Tag
hg2tag' (Hg vn vt s p) = (Tag vn' vt' s' init' aux') where
    vn' = vn
    vt' = vt
    s' = s
    init' = [Tree [([], NTNode s AA True), ([1], EmptyNode)]]
    aux' = map generateAux p
```

Note that the production $A \rightarrow C_{i,j}\sigma_1, \sigma_2$ can have the form $A \rightarrow C_{1,2}\sigma_1, \sigma_2$ or $A \rightarrow C_{2,2}\sigma_1, \sigma_2$, while j denotes the number of resulting symbols and i denotes the location where wrapping can happen. Our productions have now one of these forms, with $A, \sigma_1, \sigma_2 \in V_N$:

- $A \rightarrow C_{1,2}\sigma_1, \sigma_2$
- $A \rightarrow C_{2,2}\sigma_1, \sigma_2$
- $A \rightarrow W(\sigma_1, \sigma_2)$
- $A \rightarrow C_{1,1}(w_1 \uparrow w_2)$

Each of the productions can we converted in a specific tree.

```
generateAux :: HgProduction -> Tree
generateAux (HgProduction a (C 1 2 [HgNT b, HgNT c])) =
    Tree [
        ([], NTNode a (SA []) False),
        ([1], NTNode b AA True),
        ([1,1], NTNode a (SA []) False),
        ([2], NTNode c AA True),
        ([2,1], EmptyNode)
    ]
generateAux (HgProduction a (C 2 2 [HgNT b, HgNT c])) =
    Tree [
        ([], NTNode a (SA []) False),
        ([1], NTNode b AA True),
        ([1,1], EmptyNode),
        ([2], NTNode c AA True),
        ([2,1], NTNode a (SA []) False)
    ]
generateAux (HgProduction a (W (HgNT b, HgNT c))) =
    Tree [
        ([], NTNode a (SA []) False),
        ([1], NTNode b AA True),
        ([1,1], NTNode c AA True),
        ([1,1,1], NTNode a (SA []) False)
    ]
generateAux (HgProduction a (C 1 1 [HgT w1 w2])) =
    Tree [
        ([], NTNode a (SA []) False),
        ([1], node w1),
        ([2], NTNode a (SA []) False),
```

```

([3], node w2)
]
where
  node [w] = TNode w
  node [] = EmptyNode

```

3.4 Tag to ccg

Finally we show the algorithm to convert tree adjoining grammars to combinatory categorial grammars. Again we first convert the grammar into a normal form, after which we apply the real conversion.

```

tag2ccg :: Tag -> Ccg
tag2ccg tag = tag2ccg' (simplifyTag tag)

```

First we need to define some functions on trees. The function `isNTNode` returns `True` exactly when the given label is a non-terminal label.

```

isNTNode :: NodeLabel -> Bool
isNTNode (NTNode _ _ _) = True
isNTNode _ = False

```

All auxiliary trees have exactly one tree node labelled with a nonterminal. This node is called the foot, and its location is returned by the function `foot`.

```

foot :: Tree -> NodeLocation
foot (Tree nodes) = head [loc | (loc, lab) <- nodes, isNTNode lab,
                           length [loc' | (loc', _) <- nodes, (loc' == loc ++ [1]) ] == 0 ]

```

The path from the root of a tree to its foot is called the spine and is returned by the function with the same name.

```

spine :: Tree -> [NodeLocation]
spine = inits.foot

```

The function `addSubtree t x s` adds tree `s` at location `x` in the tree `t`.

```

addSubTree :: Tree -> NodeLocation -> Tree -> Tree
addSubTree (Tree t) x (Tree bs) = Tree (t ++ subBranches x bs)

subBranches :: NodeLocation -> [(NodeLocation, NodeLabel)] -> [(NodeLocation, NodeLabel)]
subBranches x [] = []
subBranches x ((loc, lab):bs) = [(x++loc, lab)] ++ subBranches x bs

```

`giveSubTree l t` returns the subtree of `t` with root in location `l`.

```

giveSubTree :: [Int] -> Tree -> Tree
giveSubTree loc (Tree nodes) = Tree (giveSubTree' loc nodes)

giveSubTree' :: [Int] -> [Node] -> [Node]
giveSubTree' loc [] = []
giveSubTree' loc (((loc', lab):ns)) | loc == take (length loc) loc'
    = (loc'', lab) : giveSubTree' loc ns
    | otherwise = giveSubTree' loc ns
    where loc'' = drop (length loc) loc'

```

Now we have defined these auxiliary functions, we can start to convert the tree to its normal form. In this conversion, we distinguish five different stages. For these steps, I will only give the main idea of every step, as this conversion is extensively described in the appendix of the paper by Vijay-Shanker and Weir.

```
simplifyTag :: Tag -> Tag
simplifyTag tag = tag'''' where
  (tag', saS')    = step1 tag
  tag''          = step2 tag'
  tag'''         = step3 tag'' saS'
  tag''''        = step4 tag'''
  tag''''''      = step5 tag''''
```

In the first step, all initial trees are appended to the tree given in `newAux` and added to the auxiliary trees. We also add a very simple new initial tree, consisting of a new nonterminal with the empty node as its child. Next to returning the new tag, we also return the labels of the old auxiliary trees `saS'`.

```
step1 :: Tag -> (Tag, [Int])
step1 (Tag vn vt s init aux) = ((Tag vn' vt s' init' aux'), saS') where
  vn' = vn ++ [s']
  s' = s ++ ""
  init' = [
    Tree [
      ([], NTNode s' (SA [1..n]) True),
      ([1], EmptyNode)
    ]
  ]
  aux' = aux ++ map (newAux s') init
  n = length aux'
  saS' = [k..n]
  k = (length aux)+1

newAux :: NonTerm -> Tree -> Tree
newAux s' a = addSubTree super [1] a
  where super =
    Tree [
      ([], NTNode s' (SA []) False),
      ([2], NTNode s' (SA []) False)
    ]
  ]
```

In the second step, we construct a new tag such that every tree that is labelled with a nonterminal either has a NA constraint or an OA constraint.

```
step2 :: Tag -> Tag
step2 (Tag vn vt s init aux) = (Tag vn vt s' init' aux') where
  aux' = concat (map step2newTrees aux)

step2newTrees :: Tree -> [Tree]
step2newTrees tree@(Tree nodes) = (map (Tree . (step2newTree tree)) subsets) where
  subsets = powerset ds
  ds = [n | n@(_, NTNode _ sa oa) <- nodes, sa /= (SA []), oa == False]

step2newTree :: Tree -> [Node] -> [Node]
step2newTree (Tree []) subset = []
step2newTree (Tree (n:ns)) subset = [step2newTree' n subset] ++ step2newTree (Tree ns) subset
```

```

step2newTree' :: Node -> [Node] -> Node
step2newTree' n@(location, (NTNode a sa False)) subset | (n 'elem' subset) =
    (location, NTNode a sa True)
    | otherwise =
    (location, NTNode a (SA []) False)
step2newTree' (location, 1) subset =
    (location, 1)

```

Now we convert the tag into a new tag, such that the root and foot of each auxiliary tree have an NA constraint and all other internal nodes have OA constraints with no restriction on which trees can be adjoined except that the nonterminals must match. To do this, we use all subsets of tree labels as nonterminals and include a new tree for each tree β and subsets of tree labels sa such that the label of β is an element of sa . We use the `show` function to cast the set of tree labels to a string. We use the labels of the old auxiliary trees `saS'` which we generated in step 1 as start symbol.

```

step3 :: Tag -> [Int] -> Tag
step3 (Tag vn vt s init aux) saS' = (Tag vn' vt s' init' aux') where
    vn' = map show (powerset [1..n])
    init' = [Tree [([], NTNode s' AA True), ([1], EmptyNode)]]
    aux' = step3generateAllTrees n 1 aux ++
        [Tree [([], NTNode ("[]") (SA []) False), ([1], NTNode ("[]") (SA []) False)]]
    s' = show saS'
    n = length aux

step3generateAllTrees :: Int -> Int -> [Tree] -> [Tree]
step3generateAllTrees _ _ [] = []
step3generateAllTrees g i (x:xs) = step3generateTrees g i x ++ step3generateAllTrees g (i+1) xs

step3generateTrees :: Int -> Int -> Tree -> [Tree]
step3generateTrees g betaId (Tree beta) = map (step3generateTree subTree) sa
    where
        subTree = Tree (step3generateSubTree beta)
        sa = [sa | sa <- powerset [1..g], betaId 'elem' sa]

step3generateTree :: Tree -> [Int] -> Tree
step3generateTree subTree sa = addSubTree superTree [1] subTree where
    superTree = Tree [([], (NTNode (show sa) (SA []) False)),
        ([1] ++ foot subTree ++ [1], (NTNode (show sa) (SA []) False))]

step3generateSubTree :: [Node] -> [Node]
step3generateSubTree [] = []
step3generateSubTree (t:ts) = [step3generateSubTreeNode t] ++ step3generateSubTree ts

step3generateSubTreeNode :: Node -> Node
step3generateSubTreeNode (1, (NTNode a sa oa)) = (1, (NTNode (show sa) AA oa))
step3generateSubTreeNode n = n

```

In the next stage, we 'prune' a tree by removing subtrees rooted at siblings of the spine and arranging that a simulated substitution of the removed subtree can occur. The removed subtree will be turned into an auxiliary tree and itself be pruned.

```

step4 :: Tag -> Tag
step4 (Tag vn vt s init aux) = (Tag vn' vt s init' aux') where

```

```

vn' = step4AllNonTerminals aux'
aux' = newTrees 1 aux
  where newTrees _ [] = []
        newTrees i (t:ts) = pruneTree i t ++ newTrees (i+1) ts

pruneTree :: Int -> Tree -> [Tree]
pruneTree i tree@(Tree nodes) = [Tree (concat (map (pruneNode i tree) nodes))] ++
                                concat (map (step4NewTrees i tree) nodes)

pruneNode :: Int -> Tree -> Node -> [Node]
pruneNode i tree d@(loc,lab) | loc == 'elem' (spine tree) = [d]
                             | init loc == 'elem' (spine tree) =
    [(loc, NTNode ("(" ++ show i ++ "," ++ show d ++ "))" AA True), (loc ++ [1], EmptyNode)]
                             | init (init loc) == 'elem' (spine tree) && last loc == 1
                             = [(loc, EmptyNode)]
                             | otherwise = []

step4NewTrees :: Int -> Tree -> Node -> [Tree]
step4NewTrees i tree d@(loc,lab)
  | not (loc == 'elem' (spine tree)) && (init loc) == 'elem' (spine tree) = newTree lab
  | otherwise = []
  where newTree (TNode w) =
    [Tree [
      ([], NTNode ("(" ++ show i ++ "," ++ show d ++ "))" (SA []) False),
      ([1], TNode w),
      ([2], NTNode ("(" ++ show i ++ "," ++ show d ++ "))" (SA []) False)
    ]
    newTree EmptyNode =
    [Tree [
      ([], NTNode ("(" ++ show i ++ "," ++ show d ++ "))" (SA []) False),
      ([1], EmptyNode),
      ([2], NTNode ("(" ++ show i ++ "," ++ show d ++ "))" (SA []) False)
    ]
    newTree (NTNode _ _ _) = pruneTree i (step4NewTree i tree d)

step4NewTree :: Int -> Tree -> Node -> Tree
step4NewTree j beta@(Tree nodes) d@(dloc, dlab) = Tree (betad' ++
  [(d' ++ [i], NTNode ("(" ++ show j ++ "," ++ show d ++ "))" (SA []) False)])
  where d' = step4DeepestNonTerminal (Tree betad')
        i = maximum (map child betad') + 1
        child (loc, _) | (length loc > 0) && init loc == d' = last loc
                       | otherwise = 0
        (Tree betad') = addSubTree mainTree [1] (giveSubTree dloc (Tree nodes))
        mainTree = Tree [( [], NTNode ("(" ++ show j ++ "," ++ show dloc ++ "))" (SA []) False)]

step4DeepestNonTerminal :: Tree -> NodeLocation
step4DeepestNonTerminal (Tree nodes) = foldl longest [] ntLocNodes where
  longest x y | length x >= length y = x
              | otherwise = y
  ntLocNodes = [loc | node@(loc, lab) <- nodes, isNTNode lab]

step4AllNonTerminals :: [Tree] -> [NonTerm]

```

```

step4AllNonTerminals aux =
  nub (concat [step4AllNonTerminals' n | (Tree nodes) <- aux, n <- nodes])

step4AllNonTerminals' :: Node -> [NonTerm]
step4AllNonTerminals' (_, NTreeNode nt _) = [nt]
step4AllNonTerminals' (_, TNode t)       = []
step4AllNonTerminals' (_, EmptyNode)     = []

```

Finally, we make sure that all trees are binary branching.

```

step5 :: Tag -> Tag
step5 (Tag vn vt s init aux) = (Tag vn vt s init aux') where
  aux' = map step5' aux
  step5' auxTree@(Tree nodes) =
    Tree (step5MakeTreeBinary nodes [] 1 (largestChild nodes []) [] (foot auxTree))

```

The function `step5MakeTreeBinary oldNodes oldLoc i j newLoc spine` does this job. The argument `oldNodes` consists simply of the old tree. The next three arguments define at which nodes of this tree we still have to look: only the i 'th to j 'th children of the subtree rooted in `oldLoc` are left to convert. The next argument `newLoc` is the position in the new tree we are building, and `spine` is the location to the foot node starting to look from `oldLoc`.

We recursively look at all nodes of the spine. The first case of this method is used when we look at an `oldLoc` which does not have children. The second case is for nodes with exactly one child. The third case is used when the node has more than one child, and the leftmost child is in the spine. In this case, the last child is 'split off'. The last case we use when the node has more than one child, and the leftmost child is not in the spine. Then we can 'split off' the first child. The variable `k` is the location of the node we will split off.

```

step5MakeTreeBinary :: [Node] -> NodeLocation -> Int -> Int -> [Int] -> NodeLocation -> [Node]
step5MakeTreeBinary oldNodes oldLoc i j newLoc spine
  | j == 0           = [(newLoc, currentNode)]
  | i == j           = step5MakeTreeBinary oldNodes (oldLoc ++ [head spine]) 1
    (largestChild oldNodes (oldLoc ++ [head spine])) newLoc (tail spine)
  | head spine==i    = (newLoc, currentNode) : (newLoc ++ [2], childCurrentNode) :
    (newLoc ++ [2, 1], EmptyNode) :
    step5MakeTreeBinary oldNodes oldLoc i (j-1) (newLoc ++ [1]) spine
  | otherwise        = (newLoc, currentNode) : (newLoc ++ [1], childCurrentNode) :
    (newLoc ++ [1, 1], EmptyNode) :
    step5MakeTreeBinary oldNodes oldLoc (i+1) j (newLoc ++ [2]) spine
  where currentNode = head [lab | n@(loc, lab) <- oldNodes, loc == oldLoc]
        childCurrentNode = head [lab | n@(loc, lab) <- oldNodes, loc == oldLoc ++ [k]]
        k | (head spine) == i = largestChildS oldNodes oldLoc j
          | otherwise          = i

```

This function returns the number of the largest child in nodes of node parent.

```

largestChild :: [Node] -> NodeLocation -> Int
largestChild nodes parent | children parent == [] = 0
                          | otherwise             = maximum (children parent)
  where children parent =
    [last loc | (loc, lat) <- nodes, length loc > 0, parent == init loc]

```

This function returns the number of the largest child in nodes of node parent equal to or smaller than j .

```

largestChildS :: [Node] -> NodeLocation -> Int -> Int
largestChildS nodes parent m | children parent == [] = 0
                             | otherwise             = maximum (children parent)
  where children parent =
        [last loc | (loc, lat) <- nodes, length loc > 0, parent == init loc, last loc <= m]

```

Now we can start the real conversion. We add a copy of every nonterminal to the nonterminals, and mark it with a $\hat{}$. The nonterminals and start symbol can stay the same. We add a pair to the ccg function for every auxiliary tree. If we find both $f(a) = \{x\}$ and $f(a) = \{y\}$, `correctFunction` changes it into $f(a) = \{x, y\}$.

```

tag2ccg' :: Tag -> Ccg
tag2ccg' (Tag vn vt s init aux) = Ccg vn' vt' s' f' r'
  where vn' = vn ++ map (\x -> (x ++ "^")) vn
        vt' = vt
        s' = s
        f' = correctFunction (map giveCcgFunction aux) (" : vt)
        r' = giveCcgRules f' vn

correctFunction :: [CcgFunctionPair] -> [Term] -> [CcgFunctionPair]
correctFunction f terms = [CF term (concat [v | CF term' v <- f, term == term']) | term<-terms]

```

Now we give the function to convert trees into functions. If there is a terminal in the tree, we say that $f(A) = t$ where A is the root label of that tree and t that terminal. In other cases, we get $f(A) = c \cup c^\wedge$ where $c = enc(tree, foot(tree))$.

```

giveCcgFunction :: Tree -> CcgFunctionPair
giveCcgFunction tree@(Tree nodes)
  | isEmptyNodeTree = CF "" [(A giveNtNode)]
  | isTerminalTree  = CF giveTerminal [(A giveNtNode)]
  | otherwise       = CF "" [c, addHat c]
  where isEmptyNodeTree = length node1 > 0 && tNode == EmptyNode
        isTerminalTree  = length node1 > 0 && isT tNode
        tNode           = head node1
        node1           = [lab | (loc, lab) <- nodes, loc == [1]]
        isT (TNode a)  = True
        isT _          = False
        giveTerminal   = (\(TNode t) -> t) tNode
        ntNode         = head [lab | (loc, lab) <- nodes, loc == []]
        giveNtNode     = (\(NTNode x _ _) -> x) ntNode
        c              = encTree tree (foot tree)

```

Adding $\hat{}$ to a complex tree is defined as adding this $\hat{}$ to the target category.

```

addHat :: CcgForm -> CcgForm
addHat (A nt) = A (nt ++ "^")
addHat (F o a b) = F o (addHat a) b

```

This is the `enc`-function. The first four cases of second instance are for first children, the last two for second children. The first two cases are for children without a sibling. The odd cases are for nodes where adjuncting is obligatory, the even cases for nodes where it is not possible. Note that c is the encoding of the parent.

```

encTree :: Tree -> NodeLocation -> CcgForm
encTree (Tree nodes) [] | isNa node = A nt

```

```

        | otherwise = F R (A nt) (A (nt ++ "^"))
where node = head [lab | (loc, lab) <- nodes, loc == []]
      isNa (NTNode _ (SA []) False) = True
      isNa _ = False
      nt = (\(NTNode x _ _) -> x) node
encTree tree@(Tree nodes) dn
| n==1 && noD2
      F R c (A (d1nt ++ "^"))
      && not (isNa d1lab) =
| n==1 && noD2
      && (isNa d1lab) =
      c
| n==1 && d1 'elem' (spine tree) && not (isNa d2lab) && not (isNa d1lab) =
      F R (F R c (A d2nt)) (A (d1nt ++ "^"))
| n==1 && d1 'elem' (spine tree) && not (isNa d2lab) && (isNa d1lab) =
      F R c (A d2nt)
| n==2 && d2 'elem' (spine tree) && not (isNa d1lab) && not (isNa d2lab) =
      F R (F L c (A d1nt)) (A (d2nt ++ "^"))
| n==2 && d2 'elem' (spine tree) && not (isNa d1lab) && (isNa d2lab) =
      F R c (A d1nt)

where d = init dn
      n = last dn
      d1 = d ++ [1]
      d2 = d ++ [2]
      d1lab = head [lab | (loc, lab) <- nodes, loc == d1]
      d2lab = head [lab | (loc, lab) <- nodes, loc == d2]
      d1nt = (\(NTNode x _ _) -> x) d1lab
      d2nt = (\(NTNode x _ _) -> x) d2lab
      isNa (NTNode _ (SA []) False) = True
      isNa _ = False
      c = encTree tree d
      noD2 = length [lab | (loc, lab) <- nodes, loc == d2] == 0

```

Now we show how we find the productions. For every $A \in V_N$, the productions $(x/A)A \rightarrow x$ and $A(x \setminus A) \rightarrow x$ are included. Also the productions $(x/\hat{A})(\dots(\hat{A}\dots|_1z_1)|_2\dots|_iz_i) \rightarrow (\dots(x\dots|_1z_1)|_2\dots|_iz_i)$ are included for every $A \in V_N$ and $i \leq k$ such that k is the maximal arity of the formulas in the ccg function.

```

giveCcgRules :: [CcgFunctionPair] -> [NonTerm] -> [CcgRule]
giveCcgRules ccgFunction vn =
  [CcgRule R (V "x") (A a) [] | a <- vn]
  ++ [CcgRule L (V "x") (A a) [] | a <- vn]
  ++ [CcgRule R (V "x") (A (a ++ "^")) item | item <- list k, a <- vn]
where k = maximum [arity c | (CF w cs) <- ccgFunction, c <- cs]
      list 0 = [[]]
      list k = [(R, V ("z" ++ show k)) : r | r <- list (k-1)] ++ [(L, V ("z" ++ show k)) :
        r | r <- list (k-1)] ++ []
      rest = list (k-1)

arity :: CcgForm -> Int
arity (V _) = 1
arity (A _) = 1
arity (F _ a _) = 1 + arity a

```

4 Example grammars

In this section, we will give an example for each type of grammar. All examples generate the string $a^n b^n c^n d^n$. These grammars can be used to test the conversion.

4.1 Linear indexed grammar

```
lig1 :: Lig
lig1 = Lig vn vt vi s p
  where
    vn = ["S", "T"]
    vt = ["a", "b", "c", "d"]
    vi = ["1"]
    s  = "S"
    p  = [
      LPNT ("S", []) ([ST "a"], "S", ["1"], [ST "d"]),
      LPNT ("T", ["1"]) ([ST "b"], "T", [], [ST "c"]),
      LPNT ("S", []) ([], "T", [], [ST "c"]),
      LPT ("T", []) ([])
    ]
```

4.2 Combinatory categorial grammar

```
ccg1 :: Ccg
ccg1 = Ccg vn vt s f r
  where
    vn = ["S", "T", "A", "B", "D"]
    vt = ["a", "b", "c", "d"]
    s  = "S"
    f  = [
      CF "a" [F R (A "A") (A "D")],
      CF "b" [A "B"],
      CF "c" [F L (F R (F L (A "T") (A "A"))) (A "T") (A "B")],
      CF "d" [A "D"],
      CF "" [F R (A "S") (A "T"), A "T"]
    ]
    r  = [
      CcgRule R (A "S") (A "T") [(L, A "A"), (R, A "T"), (L, A "B")],
      CcgRule L (A "S") (A "A") [(R, A "D")],
      CcgRule R (A "S") (V "y") [],
      CcgRule L (A "S") (V "y") []
    ]
```

4.3 Head grammar

```
hg1 :: Hg
hg1 = Hg vn vt s p
  where
    vn = ["S", "T"]
    vt = ["a", "b", "c", "d"]
    s  = "S"
```

```

p = [
  HgProduction "S" (C 1 1 [HgT [] []]),
  HgProduction "S" (C 2 3 [HgT ["a"] [], HgNT "T", HgT ["d"] []]),
  HgProduction "T" (W (HgNT "S", HgT ["b"] ["c"]))
]

```

4.4 Tree adjoining grammar

```

tag1 :: Tag
tag1 = Tag vn vt s init aux
  where
    vn = ["S"]
    vt = ["a", "b", "c", "d"]
    s = "S"
    init = [
      Tree [
        ([], NTNode "S" (SA [1]) False),
        ([1], EmptyNode)
      ]
    ]
    aux = [
      Tree [
        ([], NTNode "S" (SA []) False),
        ([1], TNode "a"),
        ([2], NTNode "S" (SA [1]) False),
        ([2,1], TNode "b"),
        ([2,2], NTNode "S" (SA []) False),
        ([2,3], TNode "c"),
        ([3], TNode "d")
      ]
    ]

```

5 Use of the program

The program is tested in WinHugs, but should work in any Haskell interpreter or compiler. The end user is supposed to use one of the functions `hg2tag`, `tag2ccg`, `ccg2lig` or `lig2hg`. They respectively take an `hg`, `tag`, `ccg` and `lig` as input. By combining these functions, any type of grammar can be converted into one of the others.

6 Conclusion

It is indeed possible to use the procedures given in the paper by Vijay-Shanker and Weir for actual conversion between the four grammars. However, the procedures are highly inefficient. Especially the conversion from `tag` to `ccg` results even for very simple tags in huge `ccgs`. The bad time and space complexity of the procedure, and possibly its implementation, also causes problems. The inefficiency of the conversions makes conversions over more than two steps nearly impossible. The inefficiency of the procedures can be explained by the fact that the authors of the procedures did not had the actual implementation of their procedure in mind, but were just looking for the easiest way to prove the grammars

equivalent. Currently, the most problematic conversion is from tree adjoining grammars to combinatory categorial grammars.

Further research should be done how to avoid these problems. It is very likely that methods exist which are more efficient. Another possibility is to find a procedure which converts a given grammar into the most simple definition of this grammar. This should improve the computational properties of 'chaining' our conversions. It would be especially nice to establish the property

$$\text{lig2hg}(\text{cgg2lig}(\text{tag2cgg}(\text{hg2tag}(\text{hg})))) = \text{hg}.$$

References

- G. Gazdar. Applicability of indexed grammars to natural languages. In U. Reyle and C. Rohrer, editors, *Natural Language Parsing and Linguistic Theories*, pages 69–94. Reidel, Dordrecht, 1988.
- A. K. Joshi L.S. Levy and M. Takahashi. Tree adjunct grammars. *J. Comput. Syst. Sci.*, 10(1):136–163, 1975.
- C. Pollard. *Generalized Phrase Structure Grammars, head Grammars and Natural Language*. PhD thesis, Stanford University, 1984.
- M. Steedman. Combinators and grammars. In R. Oehrle E. Bach and D. Wheeler, editors, *Categorial Grammars and Natural Language Structures*, pages 417–422. Foris, Dordrecht, 1986.
- K. Vijay-Shanker and David J. Weir. The equivalence of four extensions of context-free grammars. *Mathematical Systems Theory*, 27(6):511–546, 1994.