MG TD beam parser: SWI Prolog implementation

Stabler 185a 2012-06-09 22:15

Juan Pablo Theurel http://www.behance.net/gallery/Minimal/859897

Background. The MG beam parser implemented here is restricted by SMC and SpIC$_{mrg}$, as explained in [36]. To understand how it works, it is important to understand the way 'indices' can be used to specify linear order as explained in [35], and how that idea is extended to MGs in [36]. The use of indices was inspired by [5, 37, 22] and earlier work. The use of beam parsing here was inspired by [29, 27, 28]. A clear understanding of the simplicity of MG derivation trees and of the (deterministic multi-bottom-up) transduction to derived trees emerged gradually with the work [23, 24, 25, 26, 15, 13, 3].

0 first example grammar `mg0.pl`

This is the simple grammar used for the examples in [36]. MG grammars are just lexicons, converted to a more succinct tree representation in §1, then recognized §2, and parsed in §3. These standard MGs allow selection triggered by `=f,f` and (overt, leftward, phrasal) movement triggered by `+f,-f`. But the code is designed to be as simple as possible, setting the stage for extensions and variations like those in §8 and [34].

```
1  %   File   : mg0.pl
2  %   Author : E Stabler
3  %   Updated: Mar 2012
4  % ?- lexBuild(LexT), recognize(LexT,[the,king,prefers,the,beer]).
5  % ?- lexBuild(LexT), recognize(LexT,['Sue',knows,the,king,knows,which,beer,'John',prefers]).
6
7  :- op(500, xfy, ::). % lexical items
8  :- op(500, fx, =). % for selection features
9
10 []::[='V','C'].                 []::[='V',+wh,'C'].
11 [drinks]::[='D',='D','V'].      [prefers]::[='D',='D','V'].
12 [knows]::[='C',='D','V'].       [says]::[='C',='D','V'].
13 [the]::[='N','D'].              [which]::[='N','D',-wh].
14 [king]::['N'].                  [queen]::['N'].
15 [wine]::['N'].                  [beer]::['N'].
16 ['Sue']::['D'].                 ['John']::['D'].
17
18 startCategory('C').
```

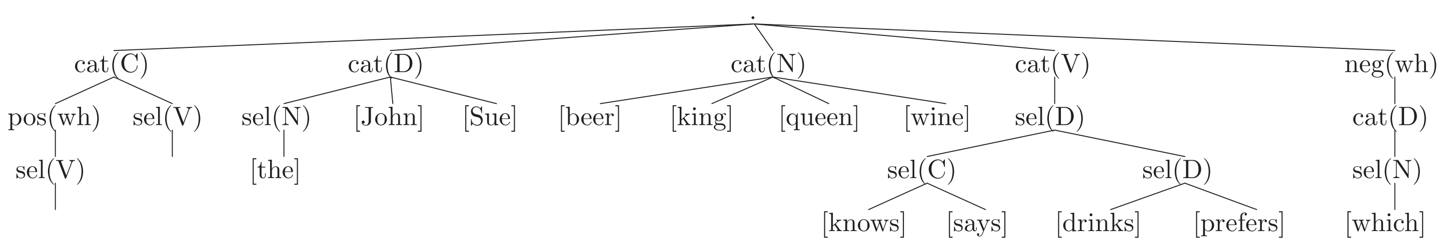
(The last example listed above, on line 5, finally gets a nice-looking derivation on page 13.)

1 lexicon builder lexBuild.pl

```
/* file: lexBuild.pl

After loading a lexicon in the original format, use this command to
build a tree representation of it.

  ?- lexBuild(T).

 */
:- op(500, xfy, ::). % lexical items
:- op(500, fx, =). % for selection features

lexBuild(T) :- bagof((S,Fs),(S::Fs),L), addLexicon(L,'.'/[],T).

addLexicon([],T,T).
addLexicon([(String,Fs)|Items],Rt/Ts0,T) :-
        rev(Fs,[String],ReverseFs),
        lexTranslate(ReverseFs,NewFs),
        addLexItem(NewFs,Ts0,Ts1), !,
        addLexicon(Items,Rt/Ts1,T).

% translate the old feature notation into the new one
lexTranslate([],[]).
lexTranslate([=F|Fs0],[sel(F)|Fs]) :- !, lexTranslate(Fs0,Fs).
lexTranslate([+F|Fs0],[pos(F)|Fs]) :- !, lexTranslate(Fs0,Fs).
lexTranslate([-F|Fs0],[neg(F)|Fs]) :- !, lexTranslate(Fs0,Fs).
lexTranslate([la(F)|Fs0],[la(F)|Fs]) :- !, lexTranslate(Fs0,Fs).
lexTranslate([ra(F)|Fs0],[ra(F)|Fs]) :- !, lexTranslate(Fs0,Fs).
lexTranslate([[]|Fs0],[[]|Fs]) :- !, lexTranslate(Fs0,Fs).
lexTranslate([[W|Ws]|Fs0],[[W|Ws]|Fs]) :- !, lexTranslate(Fs0,Fs).
lexTranslate([F|Fs0],[cat(F)|Fs]) :- lexTranslate(Fs0,Fs).

% addLexItem(Fs,Ts0,Ts) Ts is the result of making sure Fs is in Ts0,
%   adding nodes to the tree where necessary. We keep the nodes of the
%   tree in Prolog's standard alphanumeric order
addLexItem([],Ts,Ts).
addLexItem([F|Fs],[F/FTs0|Ts],[F/FTs|Ts]) :- !, addLexItem(Fs,FTs0,FTs).
addLexItem([F|Fs],[G/GTs|Ts],[F/FTs,G/GTs|Ts]) :- F @< G, !, addLexItem(Fs,[],FTs).
addLexItem([F|Fs],[T|Ts0],[T|Ts]) :- addLexItem([F|Fs],Ts0,Ts).
addLexItem([F|Fs],[],[F/FTs]) :-  addLexItem(Fs,[],FTs).

% the standard Prolog reverse
rev([],L,L).
rev([E|L0],L1,L) :- rev(L0,[E|L1],L).
```

A session using the lexical tree builder and our tree display tools:

```
Welcome to SWI-Prolog (Multi-threaded, 64 bits, Version 6.0.2)

?- [mg0,lexBuild,wish_tree].
% mg0 compiled 0.00 sec, 15 clauses
% lexBuild compiled 0.00 sec, 19 clauses
%  draw_tree compiled into draw_tree 0.00 sec, 28 clauses
%  fonttbr12 compiled into fonttbr12 0.00 sec, 233 clauses
% wish_tree compiled into wish_tree 0.00 sec, 287 clauses
true.

?- lexBuild(T), wish_tree(T).
T = '.'/[cat('C')/[pos(wh)/[sel('V')/[[]/[]]], sel('V')/[[]/[]]], cat('D')/[sel('N')/[[the]/[]]], cat('N')

?-
```



- Notice that this representation of the grammar is less redundant than the list of lexical items we started with on page 1. For example, we only write the category of `beer, kind, queen, wine` once.
- In every such tree for a lexicon containing only useful items, the nodes under the root . are always `cat(f)` or `neg(f)` for some `f`.
- In every such tree for a lexicon containing only useful items, the leaves are labeled with vocabulary sequences (of length 0 or more).
- In the parsers defined below, each predicted category will be a subtree of this lexicon, together with 0 or more subtrees representing 'movers'. To keep track of the linear order of each prediction, each subtree is associated with an index, so the predicted categories actually have the form:

  `(Tree,TreeIndex,MoverTrees,MoverTreeIndices).`

  `MoverTrees` is a list of $n \geq 0$ subtrees of the lexicon, and `MoverTreeIndices` is the list, respectively, of the n mover indices. Each of these predicted categories, as a whole, is indexed by the least element of `[TreeIndex|MoverTreeIndices]`, which just means that at each step we pop the leftmost prediction, to expand or scan it.

2 recognizer `mgbeam.pl`

We implement Stabler (2012) Appendix B, repeated here as Appendix B too.

```
% file: mgBeam.pl, implementing Stabler (2012) Appendix B
:- [library(heaps)].
% * Beam is a heap of parses, ranked by -probability, so max probability=min of heap
% * For now, probabilities of each possible expansion uniform
% * Each Parse in Beam is (Input,Q) where
% * Q is a heap of predicted items ranked by LeastIndex
% * Each predicted category in Q is (Tree,TreeI,Movers,MIndices)
% * NB pruning rule in extendBeam: improbability bound (cf Roark'01)

% INITIALIZE AND BEGIN
recognize(_/LexTs,Input) :-
    startCategory(StartF),
    memberOnce(cat(StartF)/Ts,LexTs),
    singleton_heap(Queue,[],(StartF/Ts,[],[],[])),
    singleton_heap(Beam,-1,(Input,Queue)), % -1 since Beam is a minheap
    portray_beam(Beam), % for tracing only
    extendBeam(LexTs,Beam).

% EXTEND THE BEAM RECURSIVELY
extendBeam(LexTs,Beam0) :-
    get_from_heap(Beam0,P0,(In,Q0),Beam1), % pop most probable parse
    ( success(In,Q0)
    ; get_from_heap(Q0,_,(_/Ts,TI,Movers,MIs),Q), % pop leftmost cat
      findall(Parse,(member(T,Ts),infer(T,TI,Movers,MIs,LexTs,(In,Q),Parse)),New),
      length(New,NumberOfOptions),
      ( NumberOfOptions>0,
        P is (1/NumberOfOptions)*P0, % uniform probability over next steps
        P < -0.001 -> % Simple pruning rule: improbability bound (cf Roark'01)
        insertAll(New,P,Beam1,Beam),
        portray_beam(Beam),        % for tracing only
        extendBeam(LexTs,Beam)
      ; portray_beam(Beam1), % for tracing only
        extendBeam(LexTs,Beam1)
      )
    ; empty_heap(Q0),
      portray_beam(Beam1), % for tracing only
      extendBeam(LexTs,Beam1)
    ).

% STEPS:  infer(T,I,Movers,MIs,Lex,(Input0,Queue0),(Input,Queue))
infer(Words/[],_TI,Ms,_MIs,_Lex,(In0,Q),(In,Q)) :- % SCAN
    Ms=[], append(Words,In,In0), format('~w~n',[scan:Words]).

infer(sel(F)/[FT|FTs],TI,Ms0,MIs0,LexTs,(In,Q0),(In,Q)) :-  % UNMERGE
    terminal([FT|FTs],Terminals,NonTerminals),
    append01(TI,TI0,TI1), % extend tree index TI with 0 and 1
    (  Terminals=[_|_],        % unmerge1
       memberOnce(cat(F)/CTs,LexTs),
       add_to_heap(Q0,TI0,(sel(F)/Terminals,TI0,[],[]),Q1),
       least(TI1,MIs0,Least),
       add_to_heap(Q1,Least,(cat(F)/CTs,TI1,Ms0,MIs0),Q)
    ;  NonTerminals=[_|_], % unmerge2
       memberOnce(cat(F)/CTs,LexTs),
```

```
        least(TI1,MIs0,Least),
        add_to_heap(Q0,Least,(sel(F)/NonTerminals,TI1,Ms0,MIs0),Q1),
        add_to_heap(Q1,TI0,(cat(F)/CTs,TI0,[],[]),Q)
     ;  Terminals=[_|_],   % unmerge3
        selectMI(cat(F)/CTs,OtherI,Ms0,Ms,MIs0,MIs),
        add_to_heap(Q0,TI,(sel(F)/Terminals,TI,[],[]),Q1),
        least(OtherI,MIs,Least),
        add_to_heap(Q1,Least,(cat(F)/CTs,OtherI,Ms,MIs),Q)
     ;  NonTerminals=[_|_], % unmerge4
        selectMI(cat(F)/CTs,OtherI,Ms0,Ms,MIs0,MIs),
        least(TI,MIs,Least),
        add_to_heap(Q0,Least,(sel(F)/NonTerminals,TI,Ms,MIs),Q1),
        add_to_heap(Q1,OtherI,(cat(F)/CTs,OtherI,[],[]),Q)
     ).

infer(pos(F)/[FT|FTs],TI,Ms0,MIs0,LexTs,(In,Q0),(In,Q)) :- % UNMOVE
    \+ member(neg(F)/_,Ms0),        % shortest move constraint
    ( memberOnce(neg(F)/NTs,LexTs), % unmove1
      append01(TI,TI0,TI1),
      least(TI1,[TI0|MIs0],Least),
      add_to_heap(Q0,Least,(pos(F)/[FT|FTs],TI1,[neg(F)/NTs|Ms0],[TI0|MIs0]),Q)
    ; selectMI(neg(F)/NTs,OtherI,Ms0,Ms,MIs0,MIs), % unmove2
      least(TI,[OtherI|MIs],Least),
      add_to_heap(Q0,Least,(pos(F)/[FT|FTs],TI,[neg(F)/NTs|Ms],[OtherI|MIs]),Q)
    ).

% DEFINITION OF SUCCESS: THE INPUT IS EMPTY, THE PARSE QUEUE IS EMPTY
success([],Q) :- empty_heap(Q).

% terminal(Cats,Terminals,Nontermials) split Cats into Terminals/Nonterminals
terminal([],[],[]).
terminal([Ws/[]|Ts],[Ws/[]|Trms],NonTrms) :- !, terminal(Ts,Trms,NonTrms).
terminal([T|Ts],Trms,[T|NonTrms]) :- terminal(Ts,Trms,NonTrms).

memberOnce(E,[E|_]) :- !.
memberOnce(E,[_|L]) :- memberOnce(E,L).

% insertAll(Es,P,B0,B) B is result of adding all Es to B0 with priority P
insertAll([],_,B,B).
insertAll([E|Es],P,B0,B) :- add_to_heap(B0,P,E,B1), insertAll(Es,P,B1,B).

% append(L,L0,L1) L0 is L with 0 appended, L1 has 1 appended
append01([],[0],[1]).
append01([E|L],[E|L0],[E|L1]) :- append01(L,L0,L1).

% select mover and index (note that mover is embedded!)
selectMI(E,I,[_/Ts|Es],Es,[I|Is],Is) :- member(E,Ts).
selectMI(E,I,[F|Fs],[F|Gs],[J|Is],[J|Js]) :- selectMI(E,I,Fs,Gs,Is,Js).

portray_beam(B) :- heap_to_list(B,L), heap_size(B,S),
    format('~n~w~w~n',[S,' parses in beam:']), portray_parseN(L,1).

showRootsOnly([],[]). % only roots to make trace more readable
showRootsOnly([-(P,(T/_,TI,Ms0,MsI))|L],[-(P,(T,TI,Ms,MsI))|RL]) :-
        rootsOnly(Ms0,Ms),
        showRootsOnly(L,RL).
```

```
rootsOnly([],[]).   rootsOnly([R/_|Ts],[R|Rs]) :- rootsOnly(Ts,Rs).

portray_parseN([],_). % portray each parse, numbering them from 1
portray_parseN([-(P,(In,Q))|Items],N) :-
    heap_size(Q,S), heap_to_list(Q,QL0), showRootsOnly(QL0,QL),
    format('~w~w~w~w~w~w~w~w~w~w~w~n',[N,'(',S,'). ','(',P,',(',In,',',QL,'))']),
    N1 is N+1,
    portray_parseN(Items,N1).

least(I,[],I).  % least(I,Is,J) = J is the least index among I and Is
least(I,[J|Js],Least) :- J@<I -> least(J,Js,Least); least(I,Js,Least).

% Examples
:- [pp_tree,wish_tree,lexBuild,mg0].
:- lexBuild(LexTree), wish_tree(LexTree).
:- lexBuild(LexTree), recognize(LexTree,[which,wine,the,queen,prefers]).
```

We have the following session:

```
Welcome to SWI-Prolog (Multi-threaded, 64 bits, Version 6.0.2)

?- [mgbeam].
%  library(heaps) compiled into heaps 0.00 sec, 32 clauses
%  mg0 compiled 0.00 sec, 17 clauses
%  pp_tree compiled 0.00 sec, 9 clauses
%   draw_tree compiled into draw_tree 0.00 sec, 28 clauses
%   fonttbr12 compiled into fonttbr12 0.00 sec, 233 clauses
%  wish_tree compiled into wish_tree 0.00 sec, 287 clauses
%  lexBuild compiled 0.00 sec, 19 clauses
. /[
    cat(C) /[
        pos(wh) /[
            sel(V) /[
                [] /[]]],
        sel(V) /[
            [] /[]]],
    cat(D) /[
        sel(N) /[
            [the] /[]],
        [John] /[],
        [Sue] /[]],
    cat(N) /[
        [beer] /[],
        [king] /[],
        [queen] /[],
        [wine] /[]],
    cat(V) /[
        sel(D) /[
            sel(C) /[
                [knows] /[],
                [says] /[]],
            sel(D) /[
                [drinks] /[],
                [prefers] /[]]]],
    neg(wh) /[
        cat(D) /[
            sel(N) /[
```

```
                [which] /[]]]]]

1 parses in beam:
1(1). (-1,([which,wine,the,queen,prefers],[[]- (C,[],[],[])]))
2 parses in beam:
1(2). (-0.5,([which,wine,the,queen,prefers],[[0]- (sel(V),[0],[],[]),[1]- (cat(V),[1],[],[])]))
2(1). (-0.5,([which,wine,the,queen,prefers],[[0]- (pos(wh),[1],[neg(wh)],[[0]])]))
2 parses in beam:
1(1). (-0.5,([which,wine,the,queen,prefers],[[1]- (cat(V),[1],[],[])]))
2(1). (-0.5,([which,wine,the,queen,prefers],[[0]- (pos(wh),[1],[neg(wh)],[[0]])]))
2 parses in beam:
1(2). (-0.5,([which,wine,the,queen,prefers],[[1,0]- (cat(D),[1,0],[],[]),[1,1]- (sel(D),[1,1],[],[])]))
2(1). (-0.5,([which,wine,the,queen,prefers],[[0]- (pos(wh),[1],[neg(wh)],[[0]])]))
2 parses in beam:
1(3). (-0.5,([which,wine,the,queen,prefers],[[1,0,0]- (sel(N),[1,0,0],[],[]),[1,0,1]- (cat(N),[1,0,1],[],[
2(1). (-0.5,([which,wine,the,queen,prefers],[[0]- (pos(wh),[1],[neg(wh)],[[0]])]))
1 parses in beam:
1(1). (-0.5,([which,wine,the,queen,prefers],[[0]- (pos(wh),[1],[neg(wh)],[[0]])]))
1 parses in beam:
1(2). (-0.5,([which,wine,the,queen,prefers],[[0]- (cat(V),[1,1],[neg(wh)],[[0]]),[1,0]- (sel(V),[1,0],[],[
2 parses in beam:
1(3). (-0.25,([which,wine,the,queen,prefers],[[0]- (cat(D),[0],[],[]),[1,0]- (sel(V),[1,0],[],[]),[1,1]- (
2(3). (-0.25,([which,wine,the,queen,prefers],[[0]- (sel(D),[1,1,1],[neg(wh)],[[0]]),[1,0]- (sel(V),[1,0],[
2 parses in beam:
1(4). (-0.25,([which,wine,the,queen,prefers],[[0,0]- (sel(N),[0,0],[],[]),[0,1]- (cat(N),[0,1],[],[]),[1,0
2(3). (-0.25,([which,wine,the,queen,prefers],[[0]- (sel(D),[1,1,1],[neg(wh)],[[0]]),[1,0]- (sel(V),[1,0],[
2 parses in beam:
1(3). (-0.25,([wine,the,queen,prefers],[[0,1]- (cat(N),[0,1],[],[]),[1,0]- (sel(V),[1,0],[],[]),[1,1]- (se
2(3). (-0.25,([which,wine,the,queen,prefers],[[0]- (sel(D),[1,1,1],[neg(wh)],[[0]]),[1,0]- (sel(V),[1,0],[
2 parses in beam:
1(2). (-0.25,([the,queen,prefers],[[1,0]- (sel(V),[1,0],[],[]),[1,1]- (sel(D),[1,1],[],[])]))
2(3). (-0.25,([which,wine,the,queen,prefers],[[0]- (sel(D),[1,1,1],[neg(wh)],[[0]]),[1,0]- (sel(V),[1,0],[
2 parses in beam:
1(1). (-0.25,([the,queen,prefers],[[1,1]- (sel(D),[1,1],[],[])]))
2(3). (-0.25,([which,wine,the,queen,prefers],[[0]- (sel(D),[1,1,1],[neg(wh)],[[0]]),[1,0]- (sel(V),[1,0],[
3 parses in beam:
1(3). (-0.25,([which,wine,the,queen,prefers],[[0]- (sel(D),[1,1,1],[neg(wh)],[[0]]),[1,0]- (sel(V),[1,0],[
2(2). (-0.125,([the,queen,prefers],[[1,1,0]- (sel(C),[1,1,0],[],[]),[1,1,1]- (cat(C),[1,1,1],[],[])]))
3(2). (-0.125,([the,queen,prefers],[[1,1,0]- (sel(D),[1,1,0],[],[]),[1,1,1]- (cat(D),[1,1,1],[],[])]))
5 parses in beam:
1(2). (-0.125,([the,queen,prefers],[[1,1,0]- (sel(C),[1,1,0],[],[]),[1,1,1]- (cat(C),[1,1,1],[],[])]))
2(2). (-0.125,([the,queen,prefers],[[1,1,0]- (sel(D),[1,1,0],[],[]),[1,1,1]- (cat(D),[1,1,1],[],[])]))
3(4). (-0.08333333333333333,([which,wine,the,queen,prefers],[[0]- (cat(C),[1,1,1,1],[neg(wh)],[[0]]),[1,0]
4(4). (-0.08333333333333333,([which,wine,the,queen,prefers],[[0]- (cat(D),[1,1,1,1],[neg(wh)],[[0]]),[1,0]
5(4). (-0.08333333333333333,([which,wine,the,queen,prefers],[[0]- (cat(D),[0],[],[]),[1,0]- (sel(V),[1,0],
4 parses in beam:
1(2). (-0.125,([the,queen,prefers],[[1,1,0]- (sel(D),[1,1,0],[],[]),[1,1,1]- (cat(D),[1,1,1],[],[])]))
2(4). (-0.08333333333333333,([which,wine,the,queen,prefers],[[0]- (cat(C),[1,1,1,1],[neg(wh)],[[0]]),[1,0]
3(4). (-0.08333333333333333,([which,wine,the,queen,prefers],[[0]- (cat(D),[1,1,1,1],[neg(wh)],[[0]]),[1,0]
4(4). (-0.08333333333333333,([which,wine,the,queen,prefers],[[0]- (cat(D),[0],[],[]),[1,0]- (sel(V),[1,0],
3 parses in beam:
1(4). (-0.08333333333333333,([which,wine,the,queen,prefers],[[0]- (cat(C),[1,1,1,1],[neg(wh)],[[0]]),[1,0]
2(4). (-0.08333333333333333,([which,wine,the,queen,prefers],[[0]- (cat(D),[1,1,1,1],[neg(wh)],[[0]]),[1,0]
3(4). (-0.08333333333333333,([which,wine,the,queen,prefers],[[0]- (cat(D),[0],[],[]),[1,0]- (sel(V),[1,0],
3 parses in beam:
1(5). (-0.08333333333333333,([which,wine,the,queen,prefers],[[0]- (cat(V),[1,1,1,1,1],[neg(wh)],[[0]]),[1,
```

```
2(4). (-0.08333333333333333,([which,wine,the,queen,prefers],[[0]- (cat(D),[1,1,1,1],[neg(wh)],[[0]]),[1,0]
3(4). (-0.08333333333333333,([which,wine,the,queen,prefers],[[0]- (cat(D),[0],[],[]),[1,0]- (sel(V),[1,0],
4 parses in beam:
1(4). (-0.08333333333333333,([which,wine,the,queen,prefers],[[0]- (cat(D),[1,1,1,1],[neg(wh)],[[0]]),[1,0]
2(4). (-0.08333333333333333,([which,wine,the,queen,prefers],[[0]- (cat(D),[0],[],[]),[1,0]- (sel(V),[1,0],
3(6). (-0.041666666666666664,([which,wine,the,queen,prefers],[[0]- (sel(D),[1,1,1,1,1,1],[neg(wh)],[[0]]),
4(6). (-0.041666666666666664,([which,wine,the,queen,prefers],[[0]- (cat(D),[0],[],[]),[1,0]- (sel(V),[1,0]
4 parses in beam:
1(5). (-0.08333333333333333,([which,wine,the,queen,prefers],[[0]- (cat(N),[1,1,1,1,1],[neg(wh)],[[0]]),[1,
2(4). (-0.08333333333333333,([which,wine,the,queen,prefers],[[0]- (cat(D),[0],[],[]),[1,0]- (sel(V),[1,0],
3(6). (-0.041666666666666664,([which,wine,the,queen,prefers],[[0]- (sel(D),[1,1,1,1,1,1],[neg(wh)],[[0]]),
4(6). (-0.041666666666666664,([which,wine,the,queen,prefers],[[0]- (cat(D),[0],[],[]),[1,0]- (sel(V),[1,0]
3 parses in beam:
1(4). (-0.08333333333333333,([which,wine,the,queen,prefers],[[0]- (cat(D),[0],[],[]),[1,0]- (sel(V),[1,0],
2(6). (-0.041666666666666664,([which,wine,the,queen,prefers],[[0]- (sel(D),[1,1,1,1,1,1],[neg(wh)],[[0]]),
3(6). (-0.041666666666666664,([which,wine,the,queen,prefers],[[0]- (cat(D),[0],[],[]),[1,0]- (sel(V),[1,0]
3 parses in beam:
1(5). (-0.08333333333333333,([which,wine,the,queen,prefers],[[0,0]- (sel(N),[0,0],[],[]),[0,1]- (cat(N),[0
2(6). (-0.041666666666666664,([which,wine,the,queen,prefers],[[0]- (sel(D),[1,1,1,1,1,1],[neg(wh)],[[0]]),
3(6). (-0.041666666666666664,([which,wine,the,queen,prefers],[[0]- (cat(D),[0],[],[]),[1,0]- (sel(V),[1,0]
3 parses in beam:
1(4). (-0.08333333333333333,([wine,the,queen,prefers],[[0,1]- (cat(N),[0,1],[],[]),[1,0]- (sel(V),[1,0],[]
2(6). (-0.041666666666666664,([which,wine,the,queen,prefers],[[0]- (sel(D),[1,1,1,1,1,1],[neg(wh)],[[0]]),
3(6). (-0.041666666666666664,([which,wine,the,queen,prefers],[[0]- (cat(D),[0],[],[]),[1,0]- (sel(V),[1,0]
3 parses in beam:
1(3). (-0.08333333333333333,([the,queen,prefers],[[1,0]- (sel(V),[1,0],[],[]),[1,1,0]- (cat(D),[1,1,0],[],
2(6). (-0.041666666666666664,([which,wine,the,queen,prefers],[[0]- (sel(D),[1,1,1,1,1,1],[neg(wh)],[[0]]),
3(6). (-0.041666666666666664,([which,wine,the,queen,prefers],[[0]- (cat(D),[0],[],[]),[1,0]- (sel(V),[1,0]
3 parses in beam:
1(2). (-0.08333333333333333,([the,queen,prefers],[[1,1,0]- (cat(D),[1,1,0],[],[]),[1,1,1]- (sel(D),[1,1,1]
2(6). (-0.041666666666666664,([which,wine,the,queen,prefers],[[0]- (sel(D),[1,1,1,1,1,1],[neg(wh)],[[0]]),
3(6). (-0.041666666666666664,([which,wine,the,queen,prefers],[[0]- (cat(D),[0],[],[]),[1,0]- (sel(V),[1,0]
3 parses in beam:
1(3). (-0.08333333333333333,([the,queen,prefers],[[1,1,0,0]- (sel(N),[1,1,0,0],[],[]),[1,1,0,1]- (cat(N),[
2(6). (-0.041666666666666664,([which,wine,the,queen,prefers],[[0]- (sel(D),[1,1,1,1,1,1],[neg(wh)],[[0]]),
3(6). (-0.041666666666666664,([which,wine,the,queen,prefers],[[0]- (cat(D),[0],[],[]),[1,0]- (sel(V),[1,0]
3 parses in beam:
1(2). (-0.08333333333333333,([queen,prefers],[[1,1,0,1]- (cat(N),[1,1,0,1],[],[]),[1,1,1]- (sel(D),[1,1,1]
2(6). (-0.041666666666666664,([which,wine,the,queen,prefers],[[0]- (sel(D),[1,1,1,1,1,1],[neg(wh)],[[0]]),
3(6). (-0.041666666666666664,([which,wine,the,queen,prefers],[[0]- (cat(D),[0],[],[]),[1,0]- (sel(V),[1,0]
3 parses in beam:
1(1). (-0.08333333333333333,([prefers],[[1,1,1]- (sel(D),[1,1,1],[],[])]))
2(6). (-0.041666666666666664,([which,wine,the,queen,prefers],[[0]- (sel(D),[1,1,1,1,1,1],[neg(wh)],[[0]]),
3(6). (-0.041666666666666664,([which,wine,the,queen,prefers],[[0]- (cat(D),[0],[],[]),[1,0]- (sel(V),[1,0]
3 parses in beam:
1(0). (-0.08333333333333333,([],[]))
2(6). (-0.041666666666666664,([which,wine,the,queen,prefers],[[0]- (sel(D),[1,1,1,1,1,1],[neg(wh)],[[0]]),
3(6). (-0.041666666666666664,([which,wine,the,queen,prefers],[[0]- (cat(D),[0],[],[]),[1,0]- (sel(V),[1,0]
% 4,758 inferences, 0.011 CPU in 0.866 seconds (1% CPU, 446635 Lips)
% mgbeam compiled 0.03 sec, 644 clauses
true.

?-
```

3 parser mgbeamP.pl

The parser `mgbeamP.pl` is a minor extension of the recognizer `mgbeam.pl`.

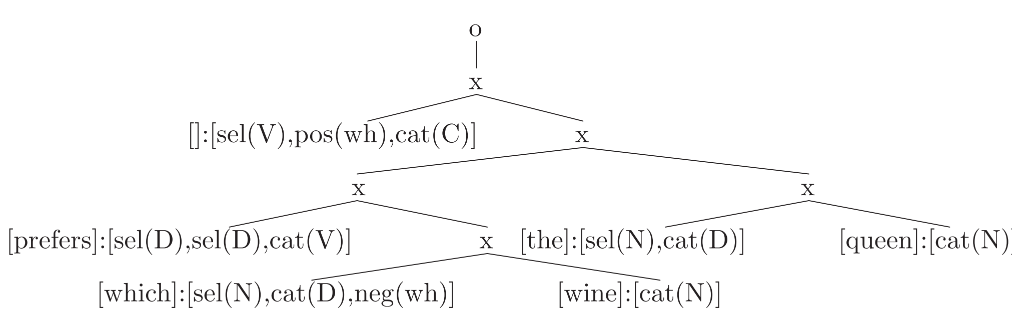
```
% file: mgBeamP.pl, implementing Stabler (2012) Appendix B
:- [library(heaps)].
% This is essentially the same as recognizer mgbeam.pl, except:
% * Each Parse in Beam is (Input,Q,D) where D is a derivation whose leaves
%   may be variables Di in predicted cats (Tree,TreeI,Ms,MIs,Di)
% * For each tree, we add a list of its ancestors in predicted cats:
%     (Tree,Anc,TreeI,Ms,Ancs,MIs,Di), now a 7-tuple, where
%  Anc is the list of ancestors of Tree in Lex, and Ancs are the ancestors of Ms

% INITIALIZE AND BEGIN
parse(_/LexTs,Input,D) :- % last arg is derivation!
    startCategory(F),
    memberOnce(cat(F)/Ts,LexTs),
    singleton_heap(Queue,[],(cat(F)/Ts,[cat(F)],[],[],[],[],DF)), % last arg is derivation!
    singleton_heap(Beam,-1,(Input,Queue,DF)),  % last arg is derivation!
%    portray_beam(Beam), % for tracing only
    extendBeam(LexTs,Beam,D).

% EXTEND THE BEAM RECURSIVELY
extendBeam(LexTs,Beam0,D) :-
    get_from_heap(Beam0,P0,(In,Q0,A0),Beam1), % pop most probable parse
    ( success(In,Q0), D=A0
    ; get_from_heap(Q0,_,(_/Ts,Anc,TI,Movers,Ancs,MIs,A),Q), % pop leftmost cat
      findall(Parse,(member(T,Ts),infer(T,Anc,TI,Movers,Ancs,MIs,A,LexTs,(In,Q,A0),Parse)),New),
      length(New,NumberOfOptions),
      ( NumberOfOptions>0,
        P is (1/NumberOfOptions)*P0, % uniform probability over next steps
        P < -0.001 -> % Simple pruning rule: improbability bound (cf Roark'01)
        insertAll(New,P,Beam1,Beam),
        %portray_beam(Beam),        % for tracing only
        extendBeam(LexTs,Beam,D)
      ; extendBeam(LexTs,Beam1,D)
      )
    ; empty_heap(Q0),  %portray_beam(Beam1), % for tracing only
      extendBeam(LexTs,Beam1,D)
    ).

% STEPS:  infer(T,I,Movers,MIs,Derivation,Lex,(Input0,Queue0),(Input,Queue))
infer(Words/[],Anc,_TI,Ms,Ancs,_MIs,(Words:Anc)/[],_Lex,(In0,Q,A),(In,Q,A)) :- % SCAN
    Ms=[], Ancs=[], append(Words,In,In0). %, format('~w~n',[scan:Words]).

infer(sel(F)/[FT|FTs],Anc,TI,Ms0,Ancs0,MIs0,x/[B,C],LexTs,(In,Q0,A),(In,Q,A)) :-  % UNMERGE
    terminal([FT|FTs],Terminals,NonTerminals),
    append01(TI,TI0,TI1), % extend tree index TI with 0 and 1
    (  Terminals=[_|_],  % unmerge1
       memberOnce(cat(F)/CTs,LexTs),
       add_to_heap(Q0,TI0,(sel(F)/Terminals,[sel(F)|Anc],TI0,[],[],[],B),Q1),
       least(TI1,MIs0,Least),
       add_to_heap(Q1,Least,(cat(F)/CTs,[cat(F)],TI1,Ms0,Ancs0,MIs0,C),Q)
    ;  NonTerminals=[_|_],  % unmerge2
       memberOnce(cat(F)/CTs,LexTs),
       least(TI1,MIs0,Least),
       add_to_heap(Q0,Least,(sel(F)/NonTerminals,[sel(F)|Anc],TI1,Ms0,Ancs0,MIs0,B),Q1),
```

```
        add_to_heap(Q1,TI0,(cat(F)/CTs,[cat(F)],TI0,[],[],[],C),Q)
     ;  Terminals=[_|_],  % unmerge3
        selectMAI(cat(F)/CTs,OtherA,OtherI,Ms0,Ms,Ancs0,Ancs,MIs0,MIs),
        add_to_heap(Q0,TI,(sel(F)/Terminals,[sel(F)|Anc],TI,[],[],[],B),Q1),
        least(OtherI,MIs,Least),
        add_to_heap(Q1,Least,(cat(F)/CTs,[cat(F)|OtherA],OtherI,Ms,Ancs,MIs,C),Q)
     ;  NonTerminals=[_|_], % unmerge4
        selectMAI(cat(F)/CTs,OtherA,OtherI,Ms0,Ms,Ancs0,Ancs,MIs0,MIs),
        least(TI,MIs,Least),
        add_to_heap(Q0,Least,(sel(F)/NonTerminals,[sel(F)|Anc],TI,Ms,Ancs,MIs,B),Q1),
        add_to_heap(Q1,OtherI,(cat(F)/CTs,[cat(F)|OtherA],OtherI,[],[],[],C),Q)
     ).

infer(pos(F)/[FT|FTs],Anc,TI,Ms0,Ancs0,MIs0,o/[B],LexTs,(In,Q0,A),(In,Q,A)) :- % UNMOVE
     \+ member(neg(F)/_,Ms0),    % shortest move constraint
     ( memberOnce(neg(F)/NTs,LexTs), % unmove1
       append01(TI,TI0,TI1),
       least(TI1,[TI0|MIs0],Least),
       add_to_heap(Q0,Least,
          (pos(F)/[FT|FTs],[pos(F)|Anc],TI1,[neg(F)/NTs|Ms0],[[neg(F)]|Ancs0],[TI0|MIs0],B),
                Q)
     ; selectMAI(neg(F)/NTs,OtherA,OtherI,Ms0,Ms,Ancs0,Ancs,MIs0,MIs), % unmove2
       least(TI,[OtherI|MIs],Least),
       add_to_heap(Q0,Least,
          (pos(F)/[FT|FTs],[pos(F)|Anc],TI,[neg(F)/NTs|Ms],[[neg(F)|OtherA]|Ancs],[OtherI|MIs],B),
                Q)
     ).

% DEFINITION OF SUCCESS: THE INPUT IS EMPTY, THE PARSE QUEUE IS EMPTY
success([],Q) :- empty_heap(Q).

% terminal(Cats,Terminals,Nontermials) split Cats into Terminals/Nonterminals
terminal([],[],[]).
terminal([Ws/[]|Ts],[Ws/[]|Trms],NonTrms) :- !, terminal(Ts,Trms,NonTrms).
terminal([T|Ts],Trms,[T|NonTrms]) :- terminal(Ts,Trms,NonTrms).

memberOnce(E,[E|_]) :- !.
memberOnce(E,[_|L]) :- memberOnce(E,L).

% insertAll(Es,P,B0,B) B is result of adding all Es to B0 with priority P
insertAll([],_,B,B).
insertAll([E|Es],P,B0,B) :- add_to_heap(B0,P,E,B1), insertAll(Es,P,B1,B).

% append(L,L0,L1) L0 is L with 0 appended, L1 has 1 appended
append01([],[0],[1]).
append01([E|L],[E|L0],[E|L1]) :- append01(L,L0,L1).

% select mover, ancestors and index (note that mover is embedded!)
selectMAI(E,A,I,[_/Ts|Es],Es,[A|As],As,[I|Is],Is) :- member(E,Ts).
selectMAI(E,A,I,[F|Fs],[F|Gs],[B|Cs],[B|Ds],[J|Is],[J|Js]) :- selectMAI(E,A,I,Fs,Gs,Cs,Ds,Is,Js).

portray_beam(B) :- heap_to_list(B,L), heap_size(B,S),
     format('~n~w~w~n',[S,' parses in beam:']), portray_parseN(L,1).

showRootsOnly([],[]). % only roots to make trace more readable
showRootsOnly([-(P,(T/_,_Anc,TI,Ms0,_Ancs,MsI,A))|L],[-(P,(T,TI,Ms,MsI,A))|RL]) :-
```

```
        rootsOnly(Ms0,Ms),
        showRootsOnly(L,RL).

rootsOnly([],[]).   rootsOnly([R/_|Ts],[R|Rs]) :- rootsOnly(Ts,Rs).

portray_parseN([],_). % portray each parse, numbering them from 1
portray_parseN([-(P,(In,Q,A))|Items],N) :-
    heap_size(Q,S), heap_to_list(Q,QL0), showRootsOnly(QL0,QL),
    format('~w~w~w~w~w~w~w~w~w~w~w~w~w~w~n',[N,'(',S,'). ','(',P,',(',In,',',QL,',',A,'))']),
    N1 is N+1,
    portray_parseN(Items,N1).

least(I,[],I).  % least(I,Is,J) = J is the least index among I and Is
least(I,[J|Js],Least) :- J@<I -> least(J,Js,Least); least(I,Js,Least).

% Tree drawing tools, lexical tree builder
:- [pp_tree,wish_tree,lexBuild,beautify,dt2bt,dt2stt,dt2xb].
:- [mg0].
%:- [mgxx].
:- [latex_tree].
%:- lexBuild(LexTree), latex_tree(LexTree), wish_tree(LexTree).
:- lexBuild(LexTree), parse(LexTree,[which,wine,the,queen,prefers],T), dt2stt(T,STT), btfy_stt(STT,B), late
%:- lexBuild(LexTree), parse(LexTree,['Sue',knows,the,king,knows,which,beer,'John',prefers],T), dt2bt(T,X)
%:- lexBuild(LexTree), parse(LexTree,['Sue',knows,the,king,knows,which,beer,'John',prefers],T), btfy(T,B),
%:- lexBuild(LexTree), parse(LexTree,['Sue',knows,the,king,knows,which,beer,'John',prefers],T), dt2bt(T,X)
%:- lexBuild(LexTree), parse(LexTree,[the,queen,prefers,the,wine],T), wish_tree(T).
%:- [mg0t0].
%:- lexBuild(LexT), parse(LexT,[these,'PL','KING','PRES','PREFER',this,'SG','BEER'],T),
%       btfy(T,D), wish_tree(D), sleep(5),
%       dt2bt(T,BT), btfy_bt(BT,BBT), wish_tree(BBT), sleep(5),
%       dt2stt(T,STT), btfy_stt(STT,BSTT), wish_tree(BSTT), sleep(5),
%       dt2xb(T,XB), btfy_xb(XB,BXB), wish_tree(BXB).
```

With this file, we have sessions like this:

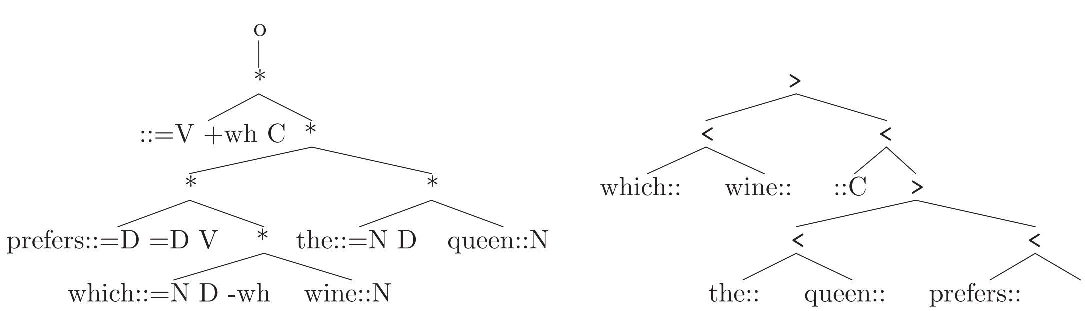
```
Welcome to SWI-Prolog (Multi-threaded, 64 bits, Version 6.0.2)

?- [mgbeamP].
%  library(heaps) compiled into heaps 0.00 sec, 32 clauses
%  mg0 compiled 0.00 sec, 17 clauses
%  pp_tree compiled 0.00 sec, 9 clauses
%   draw_tree compiled into draw_tree 0.00 sec, 28 clauses
%   fonttbr12 compiled into fonttbr12 0.00 sec, 233 clauses
%  wish_tree compiled into wish_tree 0.00 sec, 287 clauses
%  lexBuild compiled 0.00 sec, 19 clauses
% mgbeamP compiled 0.02 sec, 694 clauses
true.
```



Sacrificing the analyzability of the leaves, we can make these trees look a little more beautiful for humans with the following transducer.

```
%file: beautify.pl this transducer "beautifies" derivation trees

btfy(xx/[A,B],'**'/[BA,BB]) :- btfy(A,BA), btfy(B,BB).
btfy(x/[A,B],'*'/[BA,BB]) :- btfy(A,BA), btfy(B,BB).
btfy(o/[A],o/[BA]) :- btfy(A,BA).
btfy((Ws:Fs)/[],String/[]) :- btfyLex(Ws,Fs,String).

btfyLex(Ws,Fs,String) :-
        atomic_list_concat(Ws,' ',WsString),
        btfyFs(Fs,BFs),
        atomic_list_concat([WsString,'::'|BFs],String).

btfyFs([],[]).
btfyFs([sel(F)],['=',F]).
btfyFs([cat(F)],[F]).
btfyFs([pos(F)],['+',F]).
btfyFs([neg(F)],['-',F]).
btfyFs([ra(F)],['~r',F]).
btfyFs([la(F)],['~l',F]).
btfyFs([epp(_)],[]).
btfyFs([sel(F),G|Fs],['=',F,' '|GFs]) :- btfyFs([G|Fs],GFs).
btfyFs([cat(F),G|Fs],[F,' '|GFs]) :- btfyFs([G|Fs],GFs).
btfyFs([pos(F),G|Fs],['+',F,' '|GFs]) :- btfyFs([G|Fs],GFs).
btfyFs([neg(F),G|Fs],['-',F,' '|GFs]) :- btfyFs([G|Fs],GFs).
btfyFs([epp(_),G|Fs],GFs) :- btfyFs([G|Fs],GFs).
btfyFs([ra(F),G|Fs],['~r',F,' '|GFs]) :- btfyFs([G|Fs],GFs).
btfyFs([la(F),G|Fs],['~l',F,' '|GFs]) :- btfyFs([G|Fs],GFs).

%:- [mg0,lexBuild,wish_tree,mgbeamp,mg2dt,beautify].
%:- lexBuild(LexTree), parse(LexTree,[which,wine,the,queen,prefers],T),
%   btfy(T,B), wish_tree(B).
```

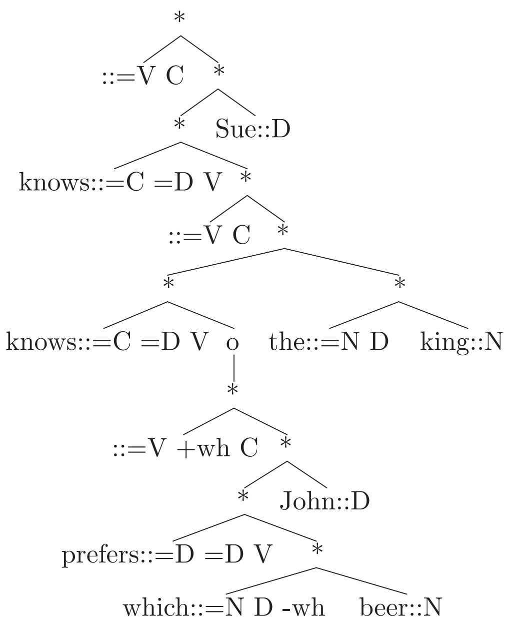
This draws the tree on the left. The tree on the right is the derived 'bare tree', which we can compute from the derivation tree using the transducer in §5 on page 17:

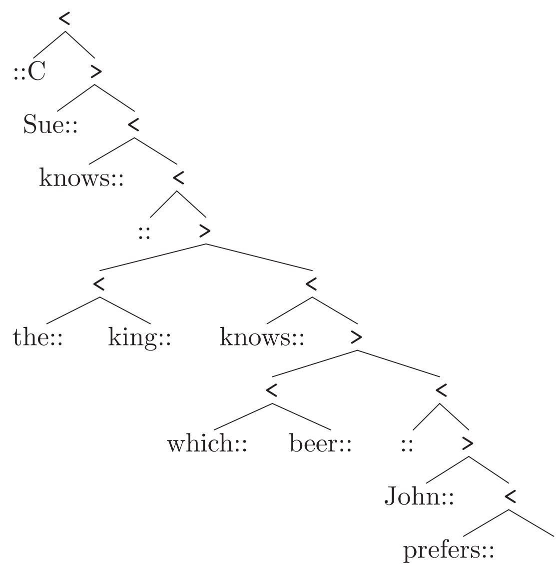


Now we can look at more complex examples much more easily. This one is defined by `mg0.pl`:

```
?- lexBuild(LexTree), parse(LexTree,['Sue',knows,the,king,knows,which,beer,'John',prefers],T),
   btfy(T,B), wish_tree(B).
```

Again, we show the 'bare tree' on the right, computed using the transducer in §5 on page 17:





Not only is the derivation tree on the left easy to look at, even though it is completely precise, but the sequence of lexical items at its leaves are in a kind of unambiguous linear order: VOS. If you take the leaves of this tree (or any other MG derivation tree), in order, you get an unambiguous Polish prefix notation, similar to writing $p \wedge (q \vee r)$ in the simpler, parenthesis-free unambiguous prefix notation $\wedge p \vee qr$. This fact is discussed in [4]. The derivation tree on the right is actually more complex.

4 transducing derivation to state dt2st.pl, dt2stt.pl

```
% file dt2st.pl  transduce derivation to state (HeadFeatures,ListOfMoverFeatures)

dt2st(x/[A/[],B],(Al,Ms)) :- %mrg1
         dt2st(B,([cat(F)],Ms)),  % B first, to determine if mrg1 is correct case
         dt2st(A/[],([sel(F)|Al],[])).
dt2st(x/[A/[A1|L1],B],(Al,Ms12)) :- %mrg2
         dt2st(B,([cat(F)],Ms2)),
         dt2st(A/[A1|L1],([sel(F)|Al],Ms1)),
         append(Ms1,Ms2,Ms12),
         smc(Ms12).
dt2st(x/[A,B],(Al,Ms12)) :- %mrg3
         dt2st(B,([cat(F),G|Be],Ms2)),
         dt2st(A,([sel(F)|Al],Ms1)),
         append(Ms1,[[G|Be]|Ms2],Ms12),
         smc(Ms12).
dt2st(xx/[A,B],(Al,Ms12)) :- %mrgEPP
        epp(F),
        dt2st(B,([cat(F),epp(F)|Be],Ms2)),
        dt2st(A,([sel(F)|Al],Ms1)),
        append(Ms1,[[neg(F)|Be]|Ms2],Ms12),
        smc(Ms12).
dt2st(o/[A],(Al,Ms)) :-
         dt2st(A,([pos(F)|Al],AMs)),
         select([neg(F)|Rest],AMs,Remainder),
         (  Rest=[] , Ms=Remainder              % move1
         ;  Rest=[_|_], Ms=[Rest|Remainder]     % move2
         ).
dt2st((_W:Fs)/[],(Fs,[])).

smc([]).
smc([[E|_]|L]) :- \+ member([E|_],L), smc(L).
```

With this file we can see the ‘state’, the features left at the end of the derivation.

```
?- lexBuild(LexTree), parse(LexTree,[which,wine,'John',prefers],T),dt2st(T,S).
LexTree = '.'/[cat('C')/[pos(wh)/[sel('V')/[[]/[]]], sel('V')/[[]/[]]], cat('D')/[sel('N')/[[the]/[]], ['J
T = o/[x/[ ([]:[sel('V'), pos(wh), cat(...)])/[], x/[x/[... / ...|...], (... : ...)/[]]]],
S = ([cat('C')], [])
```

Since the start category is C, this is an “accepting state”. This file dt2st.pl implements a finite state tree acceptor for MG derivation trees, where the states are the sequences of unchecked features.

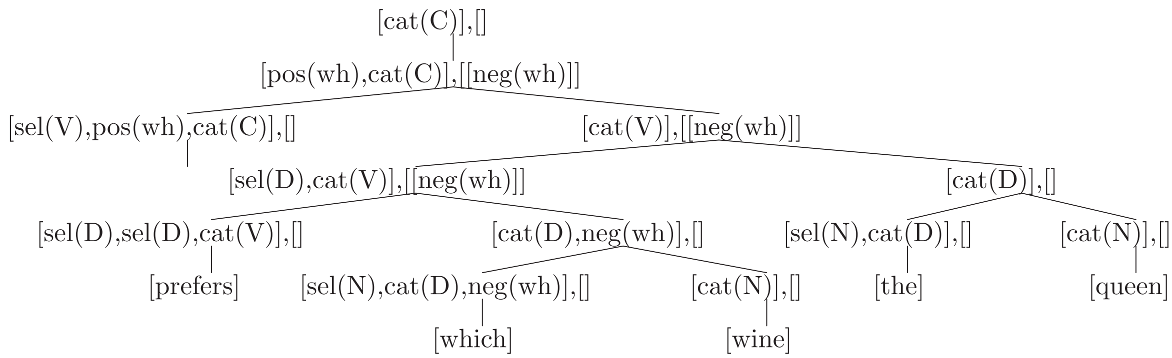
With the following file we can see the ‘state’, the features left at each step of the calculation:

```
% file dt2stt.pl  transduce derivation to state tree

dt2stt(x/[A/[],B],(Al,Ms)/[([sel(F)|Al],[])/ATs,([cat(F)],Ms)/BTs]) :- %mrg1
         dt2stt(B,([cat(F)],Ms)/BTs),  % B first, to determine if mrg1 is correct case
         dt2stt(A/[],([sel(F)|Al],[])/ATs).
dt2stt(x/[A/[A1|L1],B],(Al,Ms12)/[([sel(F)|Al],Ms1)/ATs,([cat(F)],Ms2)/BTs]) :- %mrg2
         dt2stt(B,([cat(F)],Ms2)/BTs),
         dt2stt(A/[A1|L1],([sel(F)|Al],Ms1)/ATs),
         append(Ms1,Ms2,Ms12),
         smc(Ms12).
dt2stt(x/[A,B],(Al,Ms12)/[([sel(F)|Al],Ms1)/ATs,([cat(F),G|Be],Ms2)/BTs]) :- %mrg3
         dt2stt(B,([cat(F),G|Be],Ms2)/BTs),
         dt2stt(A,([sel(F)|Al],Ms1)/ATs),
```

```
         append(Ms1,[[G|Be]|Ms2],Ms12),
         smc(Ms12).
dt2stt(xx/[A,B],(Al,Ms12)/[([sel(F)|Al],Ms1)/ATs,([cat(F)|Be],Ms2)/BTs]) :- epp(F), %mrgEPP
        dt2stt(B,([cat(F),epp(F)|Be],Ms2)/BTs),
        dt2stt(A,([sel(F)|Al],Ms1)/ATs),
        append(Ms1,[[neg(F)|Be]|Ms2],Ms12),
        smc(Ms12).

dt2stt(x/[A,B],(Al,Ms)/[([ra(F)],[])/ATs,(Al,Ms)/BTs]) :- %adjoin
         dt2stt(A,([ra(F)],[])/ATs),
         dt2stt(B,(Al,Ms)/BTs).
dt2stt(x/[A,B],(Al,Ms)/[([la(F)],[])/ATs,(Al,Ms)/BTs]) :- %adjoin
         dt2stt(A,([la(F)],[])/ATs),
         dt2stt(B,(Al,Ms)/BTs).

dt2stt(o/[A],(Al,Ms)/[([pos(F)|Al],AMs)/ATs]) :-
         dt2stt(A,([pos(F)|Al],AMs)/ATs),
         select([neg(F)|Rest],AMs,Remainder),
         (  Rest=[] , Ms=Remainder             % move1
         ;  Rest=[_|_], Ms=[Rest|Remainder]    % move2
         ).
dt2stt((W:Fs)/[],(Fs,[])/[W/[]]).


smc([]).
smc([[E|_]|L]) :- \+ member([E|_],L), smc(L).

:- [beautify].  % we use btfyFs/2 from this file

btfy_stt((Fs,Ms)/[A,B],String/[BA,BB]) :- !,
        btfyFss([Fs|Ms],BFMs),
        atomic_list_concat(BFMs,',',String),
        btfy_stt(A,BA),
        btfy_stt(B,BB).
btfy_stt((Fs,Ms)/[A],String/[BA]) :- !,
        btfyFss([Fs|Ms],BFMs),
        atomic_list_concat(BFMs,',',String),
        btfy_stt(A,BA).
btfy_stt(Ws/[],String/[]) :- atomic_list_concat(Ws,' ',String).

btfyFss([],[]).
btfyFss([Fs|More],[SFs|SMore]) :-
        btfyFs(Fs,BFs),
        atomic_list_concat(BFs,SFs),
        btfyFss(More,SMore).
```

Here is a session:

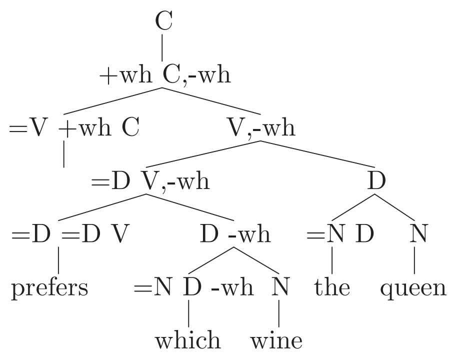
```
?- lexBuild(LexTree), parse(LexTree,[which,wine,'John',prefers],T),
   dt2stt(T,S), wish_tree(S).
```

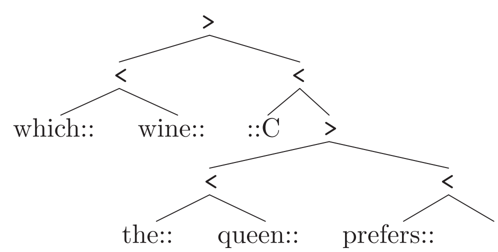


The brackets are a nuisance, so we can beautify the tree to get the one on the left below:

```
?- lexBuild(LexTree), parse(LexTree,[which,wine,'John',prefers],T),
   dt2stt(T,S), btfy_stt(S,B), wish_tree(B).
```

In addition to the beautified state tree on the left, we once again show the 'bare tree' computed in §5 on the right:





Notice that in the beautified state tree on the left, a comma separates the head feature sequence from mover feature sequences, if any (and mover sequences from each other, when constituents have more than one mover).

5 transducing to bare tree dt2bt.pl

```
% file dt2bt.pl  transduce derivation to bare tree
% Adding 2 arguments to the transducer predicates, these
% hold the head features and bare tree, respectively.

dt2bt(T,BT) :- dt2bt(T,([cat(Start)],[]),[cat(Start)],BT).

dt2bt(_/[A/[],B],(Al,Ms),Rem,'<'/[BTA,BTB]) :- %mrg1
         dt2bt(B,([cat(F)],Ms),[],BTB), % B first, to determine if mrg1 is correct case
         dt2bt(A/[],([sel(F)|Al],[]),Rem,BTA).
dt2bt(_/[A/[A1|L1],B],(Al,Ms12),Rem,'>'/[BTB,BTA]) :- %mrg2
         dt2bt(B,([cat(F)],Ms2),[],BTB),
         dt2bt(A/[A1|L1],([sel(F)|Al],Ms1),Rem,BTA),
         append(Ms1,Ms2,Ms12), smc_bt(Ms12).
dt2bt(_/[A/[],B],(Al,Ms12),Rem,'<'/[BTA,''/[]]) :- %mrg3a
         dt2bt(B,([cat(F),G|Be],Ms2),[],BTB),
         dt2bt(A/[],([sel(F)|Al],Ms1),Rem,BTA),
         append([ ([G|Be],BTB) |Ms1], Ms2 , Ms12), smc_bt(Ms12). % INSERT INTO MOVERS
dt2bt(_/[A/[A1|L1],B],(Al,Ms12),Rem,'>'/[''/[],BTA]) :- %mrg3b
         dt2bt(B,([cat(F),G|Be],Ms2),[],BTB),
         dt2bt(A/[A1|L1],([sel(F)|Al],Ms1),Rem,BTA),
         append([ ([G|Be],BTB) | Ms1], Ms2, Ms12), smc_bt(Ms12).   % INSERT INTO MOVERS

dt2bt(_/[A,B],(Al,Ms),Rem,'<'/[BTB,BTA]) :- %right adjoin
         dt2bt(A,([ra(_F)],[]),[],BTA),
         dt2bt(B,(Al,Ms),Rem,BTB).
dt2bt(_/[A,B],(Al,Ms),Rem,'>'/[BTA,BTB]) :- %left adjoin
         dt2bt(A,([la(_F)],[]),[],BTA),
         dt2bt(B,(Al,Ms),Rem,BTB).

dt2bt(o/[A],(Al,Ms),Rem,'>'/[Spec,BTA]) :-
         dt2bt(A,([pos(F)|Al],AMs),Rem,BTA),
         select( ( [neg(F)|Rest], BTB ), AMs,Remainder),  % EXTRACT FROM MOVERS
         (  Rest=[] , Ms=Remainder, Spec=BTB                    % move1
         ;  Rest=[_|_], Ms=[(Rest,BTB)|Remainder], Spec=''/[]    % move2
         ).
dt2bt(o/[A],(Al,Ms),Rem,'>'/[Spec,BTA]) :-
         dt2bt(A,([pos(F)|Al],AMs),Rem,BTA),
         select( ( [epp(F)|Rest], BTB ), AMs,Remainder),  % EXTRACT FROM MOVERS EPP case
          (  Rest=[] , Ms=Remainder, Spec=BTB                    % move1
         ;  Rest=[_|_], Ms=[(Rest,BTB)|Remainder], Spec=''/[]    % move2
         ).
dt2bt((W:Fs)/[],(Fs,[]),Rem,(W:Rem)/[]).

smc_bt([]). % the movers are now (Fs,Tree) pairs
smc_bt([([NegF|_],_Tree0)|L]) :- \+ member(([NegF|_],_Tree1),L), smc_bt(L).

:- [beautify].  % for btfyFs/2

btfy_bt(R/[A,B],R/[BA,BB]) :- !, btfy_bt(A,BA), btfy_bt(B,BB).
btfy_bt(R/[A],R/[BA]) :- !, btfy_bt(A,BA).
btfy_bt((Ws:Fs)/[],String/[]) :- !,
        atomic_list_concat(Ws,' ',WString),
        btfyFs(Fs,BFs),
        atomic_list_concat(BFs,' ',FString),
        atomic_list_concat([WString,'::',FString],String).
```

```
btfy_bt(F/[],F/[]).

btfyFss([],[]).
btfyFss([Fs|More],[SFs|SMore]) :-
        btfyFs(Fs,BFs),
        atomic_list_concat(BFs,SFs),
        btfyFss(More,SMore).

%:- [pp_tree,wish_tree,lexBuild,mg0,beautify,mgbeamp].
%:- lexBuild(LexTree), wish_tree(LexTree).
%:- lexBuild(LexTree), parse(LexTree,[which,wine,the,queen,prefers],T), dt2bt(T,B), btfy_bt(B,Y), wish_tre
```

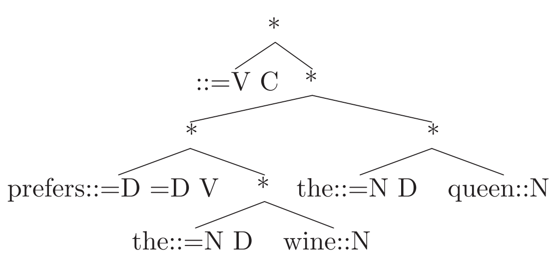
6 transducing to X' tree `dt2xb.pl`

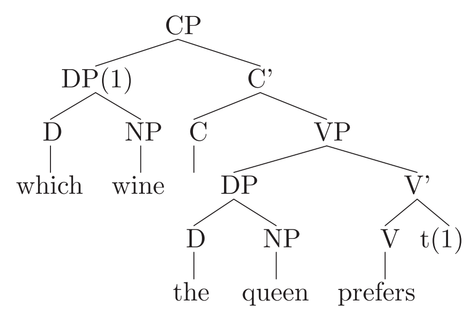
This is similar to `dt2bt.pl`.

```
% file dt2xb.pl  transduce derivation to x-bar
% Adding 2 arguments to the transducer predicates, these
% hold the head category and x-bar tree, respectively.

dt2xb(T,XP/XB) :- dt2xb(T,([cat(Cat)],[]),Cat,_/XB), catMax(Cat,XP), novars(XB).

dt2xb(_/[A/[],B],(Al,Ms),Cat,CatMid/[XBA,BP/XBB]) :- %mrg1
         dt2xb(B,([cat(F)],Ms),F,_/XBB), % B first, to determine if mrg1 is correct case
         dt2xb(A/[],([sel(F)|Al],[]),Cat,XBA),
         catMid(Cat,CatMid), catMax(F,BP).
dt2xb(_/[A/[A1|L1],B],(Al,Ms12),Cat,CatMid/[BP/XBB,XBA]) :- %mrg2
         dt2xb(B,([cat(F)],Ms2),F,_/XBB),
         dt2xb(A/[A1|L1],([sel(F)|Al],Ms1),Cat,XBA),
         append(Ms1,Ms2,Ms12), smc_xb(Ms12),  catMid(Cat,CatMid), catMax(F,BP).
dt2xb(_/[A/[],B],(Al,Ms12),Cat,CatMid/[XBA,Ti/[]]) :- %mrg3a
         dt2xb(B,([cat(F),G|Be],Ms2),F,_/XBB),
         dt2xb(A/[],([sel(F)|Al],Ms1),Cat,XBA),
         append([ ([G|Be],BPi/XBB) |Ms1], Ms2 , Ms12), smc_xb(Ms12), % INSERT INTO MOVERS
         catMid(Cat,CatMid), catMax(F,BP), BPi=..[BP,Trace], Ti=..[t,Trace].
dt2xb(_/[A/[A1|L1],B],(Al,Ms12),Cat,CatMid/[Ti/[],XBA]) :- %mrg3b
         dt2xb(B,([cat(F),G|Be],Ms2),F,_/XBB),
         dt2xb(A/[A1|L1],([sel(F)|Al],Ms1),Cat,XBA),
         append([ ([G|Be],BPi/XBB) | Ms1], Ms2, Ms12), smc_xb(Ms12),   % INSERT INTO MOVERS
         catMid(Cat,CatMid), catMax(F,BP), BPi=..[BP,Trace], Ti=..[t,Trace].

dt2xb(_/[A,B],(Al,Ms),Cat,BP/[BP/BTB,AP/BTA]) :- %right adjoin
         dt2xb(A,([ra(Cat)],[]),Cat,_/BTA),
         dt2xb(B,(Al,Ms),Cat,_/BTB),
         atomic_concat('~r',Cat,ACat),  % Since the usual adjunct cats not given
         catMax(ACat,AP), catMax(Cat,BP).
dt2xb(_/[A,B],(Al,Ms),Cat,BP/[AP/BTA,BP/BTB]) :- %left adjoin
         dt2xb(A,([la(Cat)],[]),Cat,_/BTA),
         dt2xb(B,(Al,Ms),Cat,_/BTB),
         atomic_concat('~l',Cat,ACat), % Since the usual adjunct cats not given
         catMax(ACat,AP), catMax(Cat,BP).

dt2xb(o/[A],(Al,Ms),Cat,CatMid/[Spec,XBA]) :-
         dt2xb(A,([pos(F)|Al],AMs),Cat,XBA),
         select( ( [neg(F)|Rest], XBB ), AMs,Remainder),  % EXTRACT FROM MOVERS
         (  Rest=[] , Ms=Remainder, Spec=XBB                       % move1
         ;  Rest=[_|_], Ms=[(Rest,XBB)|Remainder], Spec=''/[]     % move2
         ),
         catMid(Cat,CatMid).
dt2xb(o/[A],(Al,Ms),Cat,CatMid/[Spec,XBA]) :-
         dt2xb(A,([pos(F)|Al],AMs),Cat,XBA),
         select( ( [epp(F)|Rest], XBB ), AMs,Remainder),  % EXTRACT FROM MOVERS - EPP case
         (  Rest=[] , Ms=Remainder, Spec=XBB                       % move1
         ;  Rest=[_|_], Ms=[(Rest,XBB)|Remainder], Spec=''/[]     % move2
         ),
         catMid(Cat,CatMid).
dt2xb((W:Fs)/[],(Fs,[]),Cat,Cat/[W/[]]).

catMid(F,FBar) :- atomic_concat(F,'''',FBar).
```

```
catMax(F,FMax) :- atomic_concat(F,'P',FMax).

smc_xb([]). % movers are now (Fs,XBTree) pairs
smc_xb([([NegF|_],_Tree0)|L]) :- \+ member(([NegF|_],_Tree1),L), smc_xb(L).

btfy_xb(R/[A,B],R/[BA,BB]) :- !, btfy_xb(A,BA), btfy_xb(B,BB).
btfy_xb(R/[A],R/[BA]) :- !, btfy_xb(A,BA).
btfy_xb(t(N)/[],t(N)/[]) :- !.
btfy_xb(''/[],''/[]) :- !.
btfy_xb(Ws/[],String/[]) :- atomic_list_concat(Ws,' ',String).

novars(M) :- nvs(M,1,_). % instantiate the variables with numbers

nvs(M,M,N):- !, succ(M,N).
nvs(Term,M,N):- functor(Term,_,Arity), nvs(0,Arity,Term,M,N).

nvs(A,A,_,N,N):- !.
nvs(Am,Arity,Term,M,N) :- succ(Am,An), arg(An,Term,Arg), nvs(Arg,M,K), !, nvs(An,Arity,Term,K,N).

%:-[pp_tree,wish_tree,lexBuild,beautify].
%:-[mg0].
%:-lexBuild(LexTree),wish_tree(LexTree).
%:-lexBuild(LexTree),parse(LexTree,[which,wine,the,queen,prefers],T), btfy(T,B), wish_tree(B).
%:-lexBuild(LexTree),parse(LexTree,[which,wine,the,queen,prefers],T), dt2xb(T,S), btfy_xb(S,B), wish_tree(
```

```
?- lexBuild(LexTree), parse(LexTree,[which,wine,'John',prefers],T),
   dt2xb(T,S), btfy_xb(S,B), wish_tree(B).
```



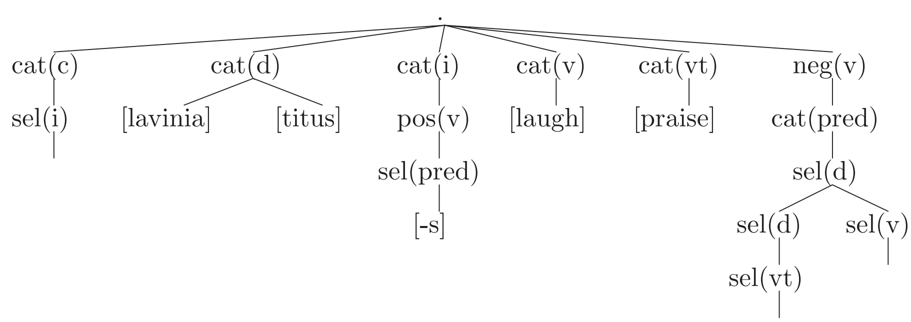


7 more example grammars

The grammars of the next 3 sections are inspired by Mahajan [21], pointing out that any constituent order can be derived from one basic underlying form, one underlying mechanism which uniformly combines (head complement) and then (specifier (head complement)). 'Antisymmetric' proposals along these lines are developed by Sportiche [30, 31], Koopman [20, 16, 17], Kayne [11, 10, 9, 8, 7] and many others; cf. Abels & Neeleman for a challenge [1].

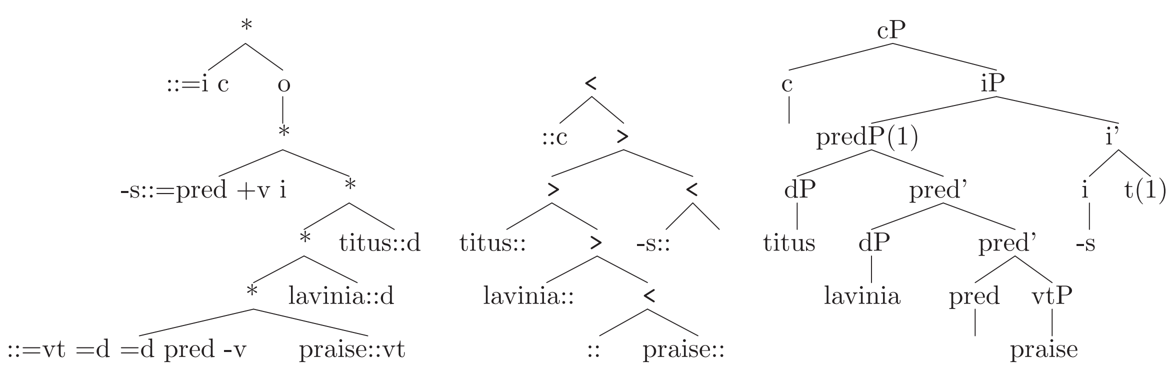
7.1 SOVI naive Tamil mg-nt.pl

```
%   File   : g-nt.pl - naive Tamil SOVI
%   Author : E Stabler
%   Updated: Mar 00

[]::[=i,c].
['-s']::[=pred,+v,i].
[]::[=vt,=d,=d,pred,-v].      []::[=v,=d,pred,-v].
[praise]::[vt].               [laugh]::[v].
[lavinia]::[d].               [titus]::[d].

startCategory(c).
```



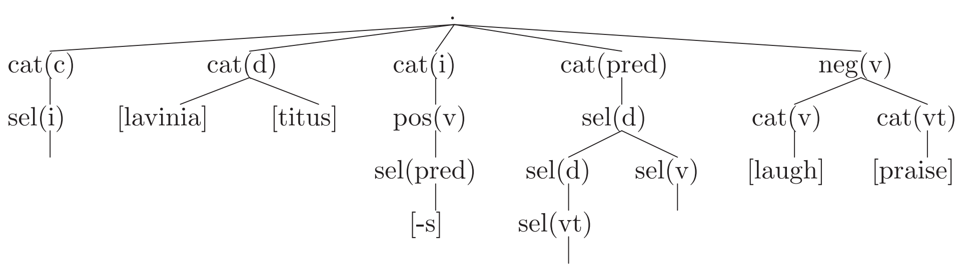

```
?- ['mg-nt'], lexBuild(LexTree), parse(LexTree,[titus,lavinia,praise,'-s'],T),
   dt2bt(T,B), btfy_bt(B,BT), wish_tree(BT).
% mg-nt compiled 0.00 sec, 2 clauses
LexTree = '.'/[cat(c)/[sel(i)/[[]/[]]], cat(d)/[[lavinia]/[], [titus]/[]], cat(i)/[pos(v)/[sel(...)/[...]]
T = x/[ ([]:[sel(i), cat(c)])/[], o/[x/[ ([...]:[...|...])/[], x/[...|...]]]],
B = (<)/[ ([]:[cat(c)])/[], (>)/[ (>)/[ ([...]:[])/[], (>)/[...|...]], (<)/[ (... : ...)/[], ... / ...]]],
BT = (<)/['::c'/[], (>)/[ (>)/['titus::'/[], (>)/[...|...]], (<)/['-s::'/[], ... / ...]]]
```

Here we show the (beautified) derivation, bare tree, and X-bar trees that our transducers compute:



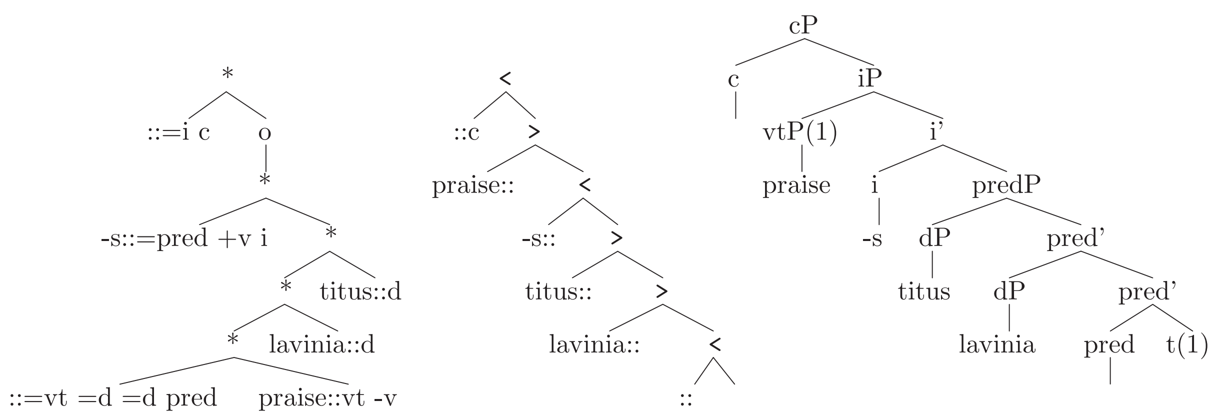
7.2 VISO naive Zapotec mg-nz.pl

```
%   File   : g-nz.pl - naive zapotec VISO
%   Author : E Stabler
%   Updated: Mar 00

[]::[=i,c].
['-s']::[=pred,+v,i].
[]::[=vt,=d,=d,pred].        []::[=v,=d,pred].
[praise]::[vt,-v].           [laugh]::[v,-v].
[lavinia]::[d].              [titus]::[d].

startCategory(c).
```



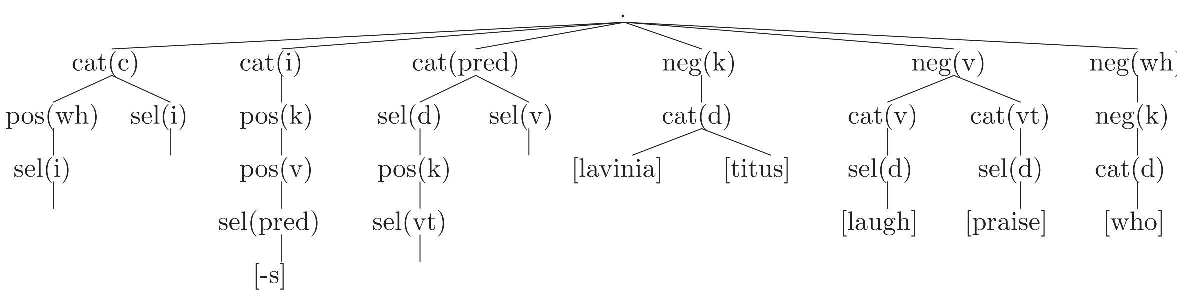

```
?- ['mg-nz'], lexBuild(LexTree), parse(LexTree,[praise,'-s',titus,lavinia],T), dt2bt(T,DT), btfy_bt(DT,BT)
% mg-nz compiled 0.00 sec, 2 clauses
LexTree = '.'/[cat(c)/[sel(i)/[[]/[]]], cat(d)/[[lavinia]/[], [titus]/[]], cat(i)/[pos(v)/[sel(...)/[...]]
T = x/[ ([]:[sel(i), cat(c)])/[], o/[x/[ ([...]:[...|...])/[], x/[...|...]]]],
DT = (<)/[ ([]:[cat(c)])/[], (>)/[ ([praise]:[])/[], (<)/[ (... : ...)/[], ... / ...]]],
BT = (<)/['::c'/[], (>)/['praise::'/[], (<)/['-s::'/[], ... / ...]]]
```

Here we show the (beautified) derivation, bare tree, and X-bar trees that our transducers compute:



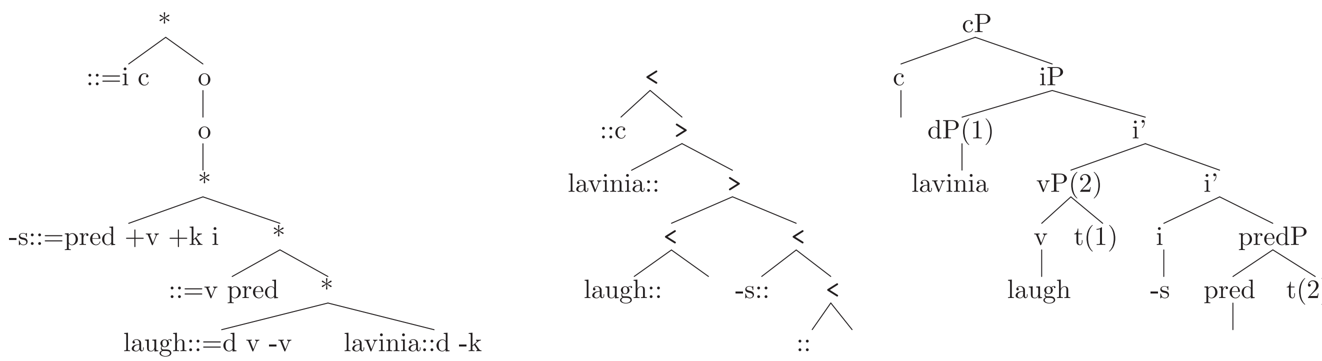
7.3 SVIO naive English `mg-ne.pl`

```
%   File   : g-ne.pl - naive english
%   Author : E Stabler
%   Updated: Mar 00

[]::[=i,c].                 []::[=i,+wh,c].
['-s']::[=pred,+v,+k,i].    []::[=vt,+k,=d,pred].    []::[=v,pred].
[praise]::[=d,vt,-v].       [laugh]::[=d,v,-v].
[lavinia]::[d,-k].          [titus]::[d,-k].         [who]::[d,-k,-wh].

startCategory(c).
```

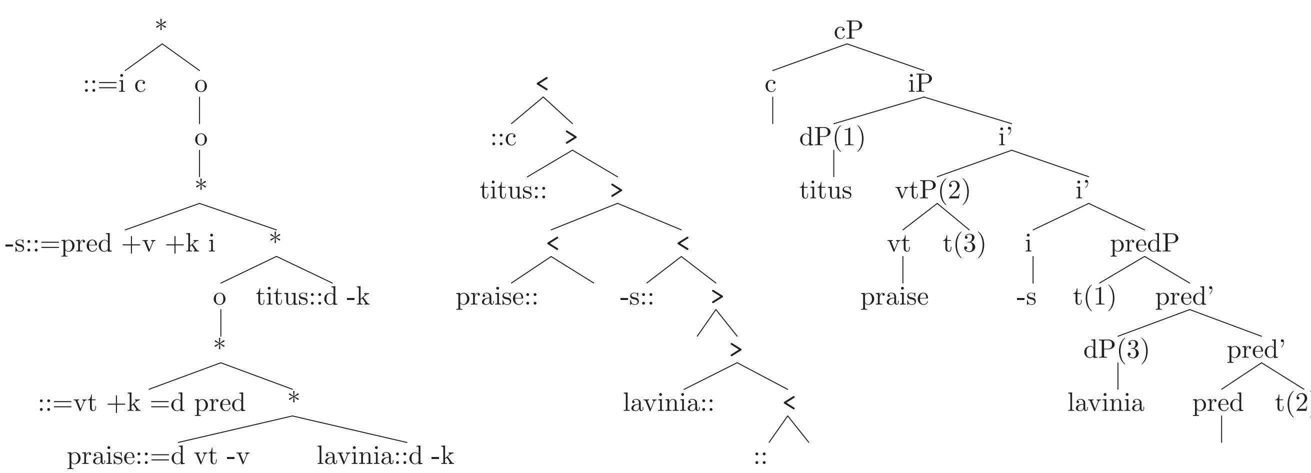



```
?- ['mg-ne'], lexBuild(L), parse(L,[lavinia,laugh,'-s'],T), btfy(T,B), wish_tree(B).
```

Here we show the (beautified) derivation, bare tree, and X-bar trees that our transducers compute:

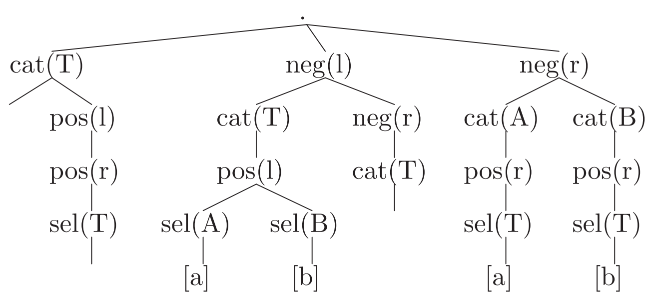



```
?- ['mg-ne'], lexBuild(L), parse(L,[titus,praise,'-s',lavinia],T),
   dt2bt(T,D), btfy_bt(D,B), wish_tree(B).
```



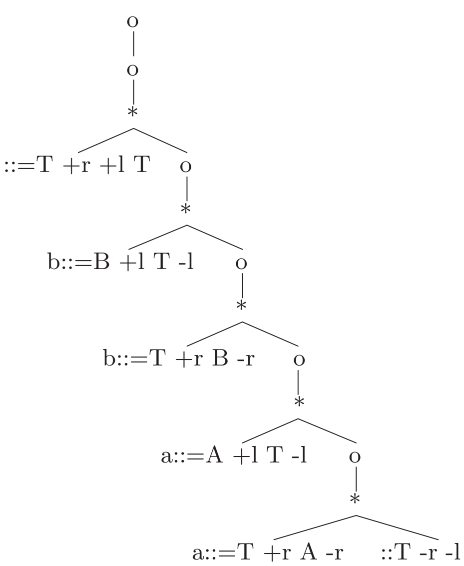
7.4 the copy language mgxx.pl

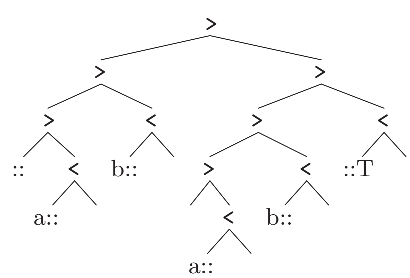
```
%   File   : mgxx.pl
%   Author : E Stabler
%   Updated: Mar 00
%     grammar for the copy language {xx| x\in{a,b}*}

[]::['T'].
[]::['T',-r,-l].               []::[='T',+r,+l,'T'].
[a]::[='T',+r,'A',-r].         [b]::[='T',+r,'B',-r].
[a]::[='A',+l,'T',-l].         [b]::[='B',+l,'T',-l].

startCategory('T').
```

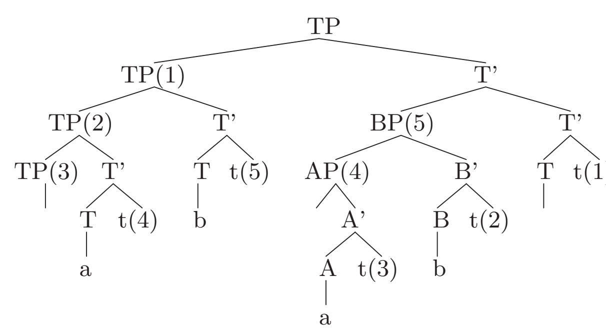



```
?- [wish_tree,lexBuild,beautify,mgbeamp,mgxx,dt2bt].
?- lexBuild(LexTree), parse(LexTree,[a,b,a,b],T), dt2bt(T,BT), btfy_bt(BT,D), wish_tree(D).
```

Here we show the (beautified) derivation, bare tree, and X-bar trees that our transducers compute:

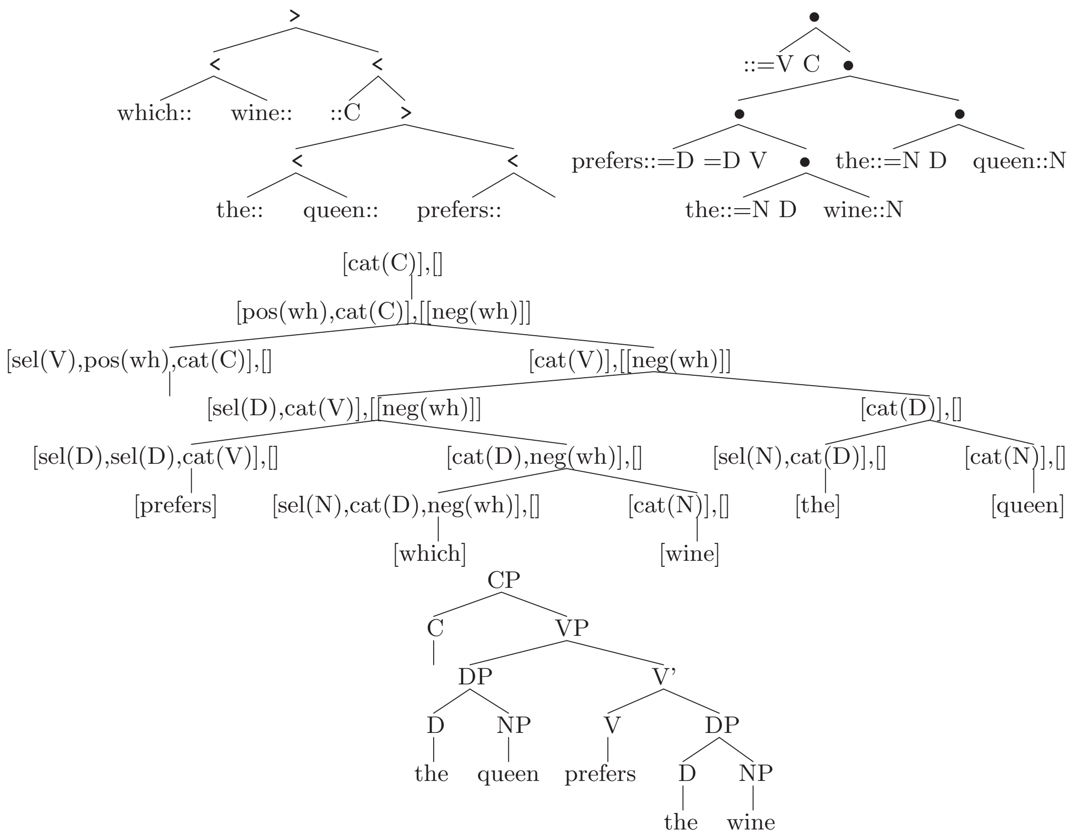






8 extensions

(1) In our first examples, we have ignored tense and agreement.



(2) In simple English sentences like *the queen prefers the wine*:

a. Verbs can show agreement with their subjects

the queen$\underline{\epsilon}$ prefers/*the queens prefers

b. Determiners can show agreement with their nouns

that queen$\underline{\epsilon}$/*that queens

c. T is above VP, as evidenced by facts like this

- In English auxiliary verb sequences, T expressed on the first verb only.

  The queen has been preferring wine.

- T can be stranded apart from the verb in questions, ellipsis, rescued by DO

  Does the queen prefer wine?/I prefer wine, and the queen does ~~prefer wine~~ too.

But obviously, if T is above the VP, and if auxiliary verbs do not take subject but can select a main verb that does have a subject, then subject of the sentence has to move from VP to above T.

d. TP, VP, NP can be modified by PP, AdvP, AP

[she prefers the wine] on Sunday, really [prefers the wine], red [wine]

e. Successive cyclic movements: In "raising constructions", a D can move to an EPP position, and then to a higher one. And in "wh movement" a wh-phrase can move to spec,C and then to a higher one.

John [seems [_ happy]] / John is [_ likely to [_ seem [_ happy]]]

Mary said [who [you saw _]] / Who did [_ Mary say [_ [you saw _]]]?

(3) One standard account gets the heads to the right places and ensures agreement with these assumptions (see e.g. §§8,12 of the text by Koopman, Sportiche & Stabler 2012):

1. extended projection principle (EPP) and successive cyclic movement:
   - T has a +epp(D[nominative]) feature, triggering movement of a lower -D
   - When a D is selected, -D can remain visible for EPP movement
   - When a -wh or -D moves, it can remain visible for another movement
2. adjunction of modifiers:
   - modifiers adjoin to XP to form an XP
3. head movement: a head adjoins to the head selecting it
   - V-to-v movement: PREFER moves up to the empty v, creating PREFER+ϵ
   - T-to-v affix hopping: PRES moves down to v to create PREFER+ϵ+-s
   - Num-to-N affix hopping: -s moves down to N to create king+-s
4. agreement
   - spec-head: the moved -epp DP agrees with the +epp T head in person, number.

     Koopman [18] argues that agreement between D and marking on the Noun may be spec-head too, but in the simple example grammar developed here, we treat this as a special case.

- Linguists are looking for simpler, more parsimonius accounts, but adopting this one, it is not too hard to add these 4 basic mechanisms to MGs. . .

8.1 EPP and persistent features

(4) Let's represent the +EPP(D) feature simply by +D, and define

A feature $+f$ is an **EPP licensor** iff f is a category in any lexical item.
In such a case, we also say epp(f) holds, and say f is an **EPP category**.

(5) We add the following selection rule, and <u>all</u> selection rules are constrained by this additional condition:

(SMC$_{EPP}$) No rule can check a category feature f if there is a $-f$ mover.

$$\frac{s \cdot {=}f\gamma, \mu_0 \qquad t \cdot f\beta, \mu_1}{s : \gamma, \mu_0 \uplus \mu_1 \uplus \{t : -f\beta\}}\ (\bullet\bullet)\text{ select epp mover, epp}(f)$$

for $\beta \in F^*$ any sequence of features, possibly empty.

Notice how the category feature f "persists" as a licensee feature even after it has been checked. And note that (SMC$_{EPP}$) does not allow a D subject to be merged into a derivation when a lower object has put a -D into the mover list.

Also notice the similarity between $\bullet_{epp}$ and the rule $\bullet_3$ that we are already using. It is no surprise that adding this rule does not affect the weak expressive power of these grammars.

However: **the union of this rule with earlier ones is not a function**.

(6) For successive cyclic movement, we add the following rule:

$$\frac{s : {+}f\gamma, \mu \uplus \{t : {-}f\beta\}}{s : \gamma, \mu \uplus \{t : {-}f\beta\}}\ (\circ\circ)\text{ nonfinal move of licensee.}$$

Notice how the licensee feature $-f$ "persists" as a licensee feature even after it has been checked. But this rule is quite similar to $\circ_2$, and so expressive power of the grammar is not affected.

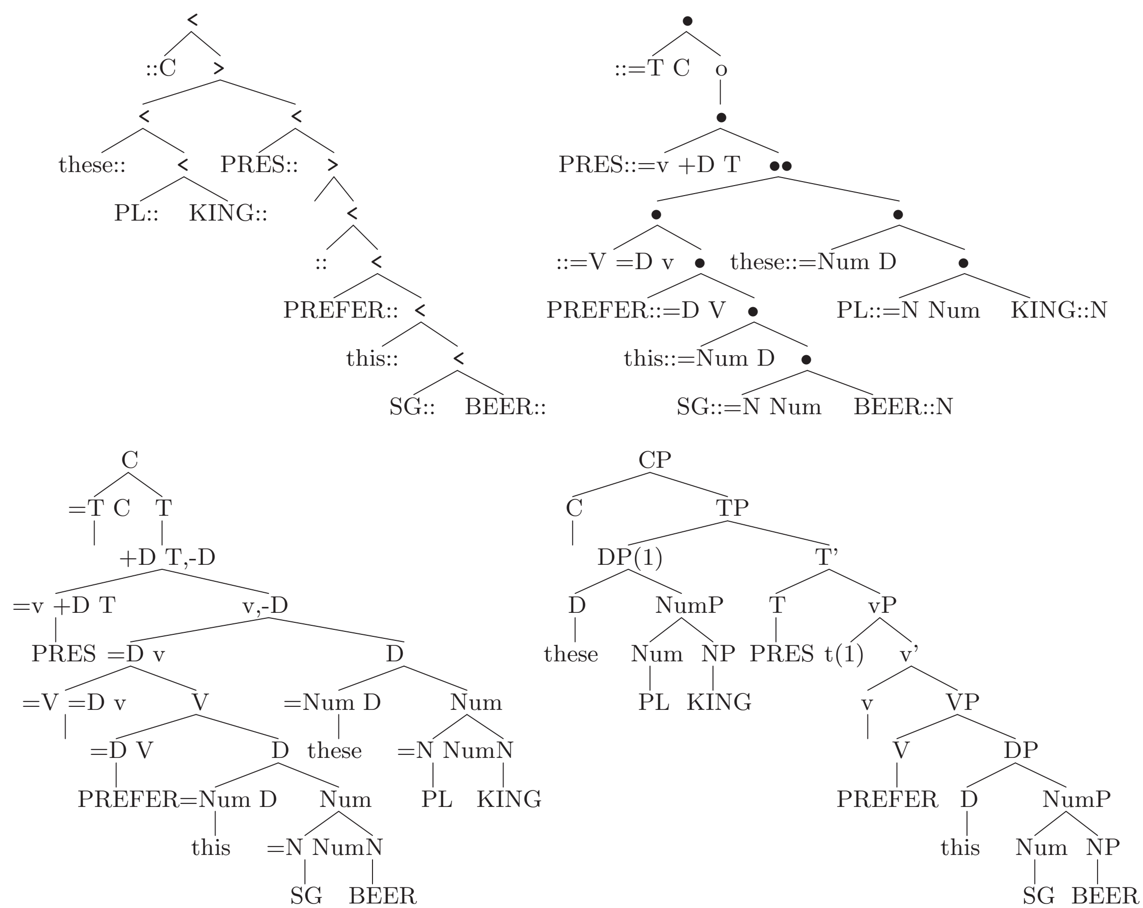
But again: **the union of this rule with earlier ones is not a function**.

(7) **Example.** With these additions, consider the following lexicon:

```
%   File   : mg0t0.pl
%   Author : E Stabler
%   Updated: June 2012
% :- lexBuild(LexT), parse(LexT,[these,'PL','KING','PRES','PREFER',this,'SG','BEER'],T),

:- op(500, xfy, ::). % lexical items
:- op(500, fx, =). % for selection features

[]::[='T','C'].            ['PRES']::[=v,+'D','T'].
[]::[='V',='D',v].         % "little v" aspect introduces the subject
['PREFER']::[='D','V'].
[this]::[='Num','D'].      [these]::[='Num','D'].
['SG']::[='N','Num'].      ['PL']::[='N','Num'].
['KING']::['N'].           ['QUEEN']::['N'].
['WINE']::['N'].           ['BEER']::['N'].

epp('D').
startCategory('C').
```

Notice that 'D' is listed as an EPP category, even though this could easily be calculated: it is EPP because we have the licensor +D in the lexical entries with category tense, T.

Using the parser of §8.3 that implements these rules, we derive:



To get these and similar trees into finished, pronounced form, we still need to deal with agreement and head movement – a standard approach is implemented in [32] and compared to some alternatives in [33]. But let's consider adjunction of modifiers first since it is easier.

8.2 adjunction

(8) Adapting Frey & Gärtner [2], for any category f we introduce the features $\approx_l f, \approx_r f$ which intuitively mean "adjoin me to the left,right (respectively) of f to produce an f." For these new adjunction features, we add these rules:

$$\frac{s \cdot \approx_l f, \emptyset \qquad t \cdot f\beta, \mu}{st : f\beta, \mu}\ (\approx_l)\ \text{left adjoin}$$

$$\frac{s \cdot \approx_l f\delta, \emptyset \qquad t \cdot f\beta, \mu}{t : f\beta, \mu \uplus \{s : \delta\}}\ (\approx_l)\ \text{left adjoin mover}$$

$$\frac{s \cdot \approx_r f, \emptyset \qquad t \cdot f\beta, \mu}{ts : f\beta, \mu}\ (\approx_r)\ \text{right adjoin}$$

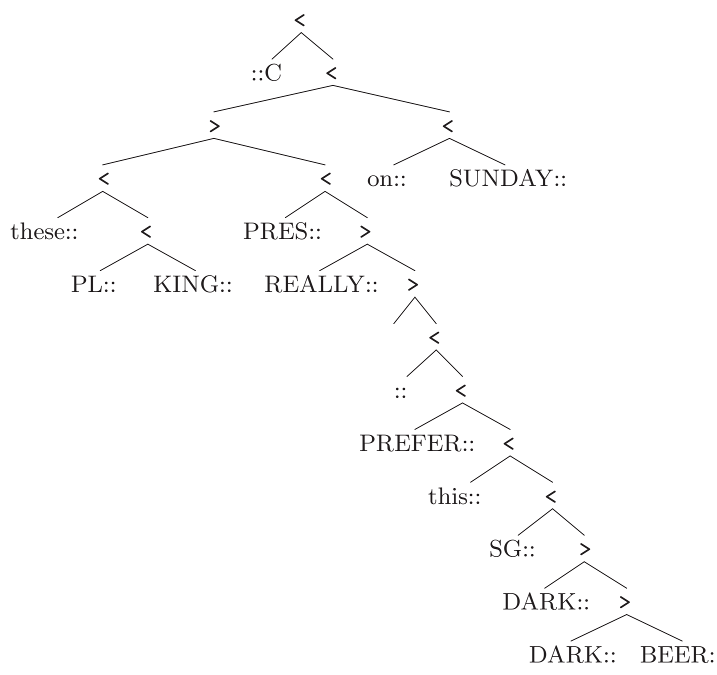
$$\frac{s \cdot \approx_r f, \emptyset \qquad t \cdot f\beta, \mu}{ts : f\beta, \mu \uplus \{s : \delta\}}\ (\approx_r)\ \text{right adjoin mover}$$

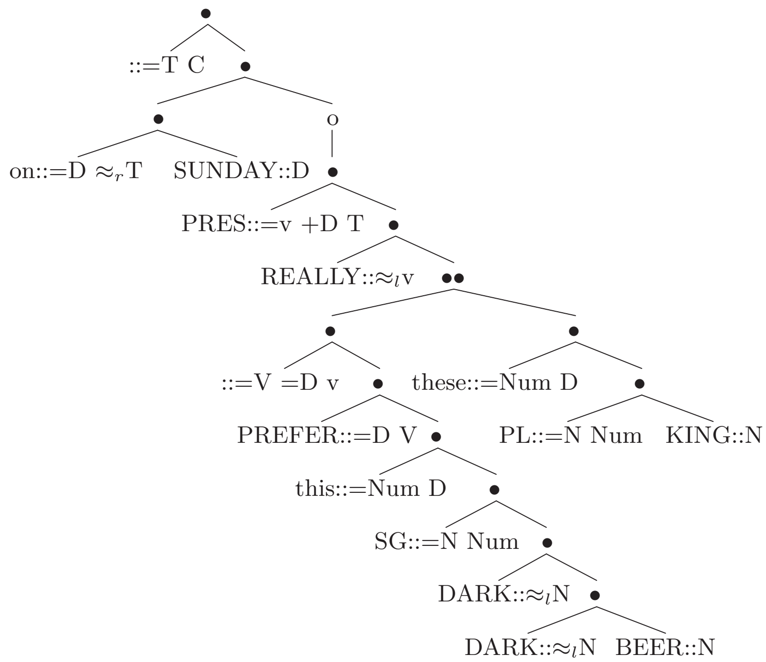
Note that these rules do not allow adjuncts to have any movers, enforcing the "Adjunct Island" condition. Also notice the similarity between these new rules and $\bullet_1, \bullet_2$, respectively, which we were already using. It is no surprise that adding these new rules does not affect the weak expressive power of these grammars.

(9) **Example.** With these additions, consider the following lexicon, writing `la(f)` for $\approx_l f$ and `ra(f)` for $\approx_r f$:

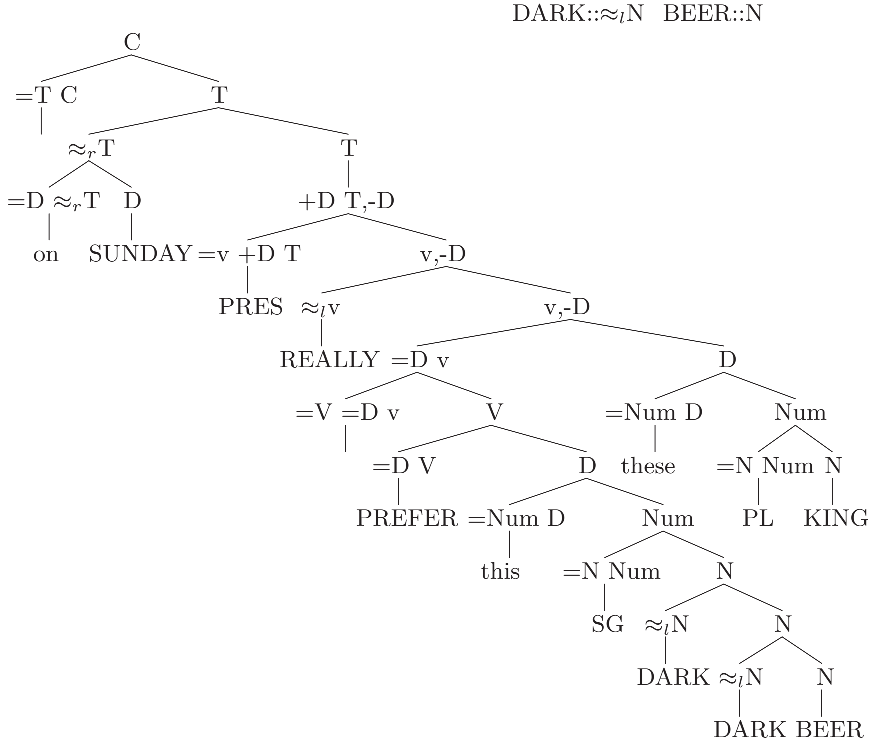
```
%   File   : mg0t1.pl
%   Author : E Stabler
%   Updated: June 2012
% :- lexBuild(L), parse(L,[these,'PL','KING','PRES','PREFER',this,'SG','DARK','BEER'],T),
:- op(500, xfy, ::). % lexical items
:- op(500, fx, =). % for selection features

[]::['T','C'].            []::[='T',+wh,'C'].      ['PRES']::[=v,+'D','T'].
[]::[='V',='D',v].        % "little v" introduces the subject
['PREFER']::[='D','V'].   ['KNOW']::[='C','V'].    ['LAUGH']::['V'].
[this]::[='Num','D'].     [these]::[='Num','D'].   [which]::[='Num','D',-wh].
['SG']::[='N','Num'].     ['PL']::[='N','Num'].
['KING']::['N'].          ['QUEEN']::['N'].        ['WINE']::['N'].     ['BEER']::['N'].
['MARY']::['D'].          ['JOHN']::['D'].         ['DENMARK']::['D'].  ['SUNDAY']::['D'].

['REALLY']::[la(v)].
['RED']::[la('N')].       ['WHITE']::[la('N')].     ['DARK']::[la('N')].
[with]::[='D',ra('N')].   [from]::[='D',ra('N')].
[on]::[='D',la('T')].     [on]::[='D',ra('T')].     [tomorrow]::[ra('T')].

epp('D').  startCategory('C').
```

Using the parser of §8.3 that implements these rules, we derive:

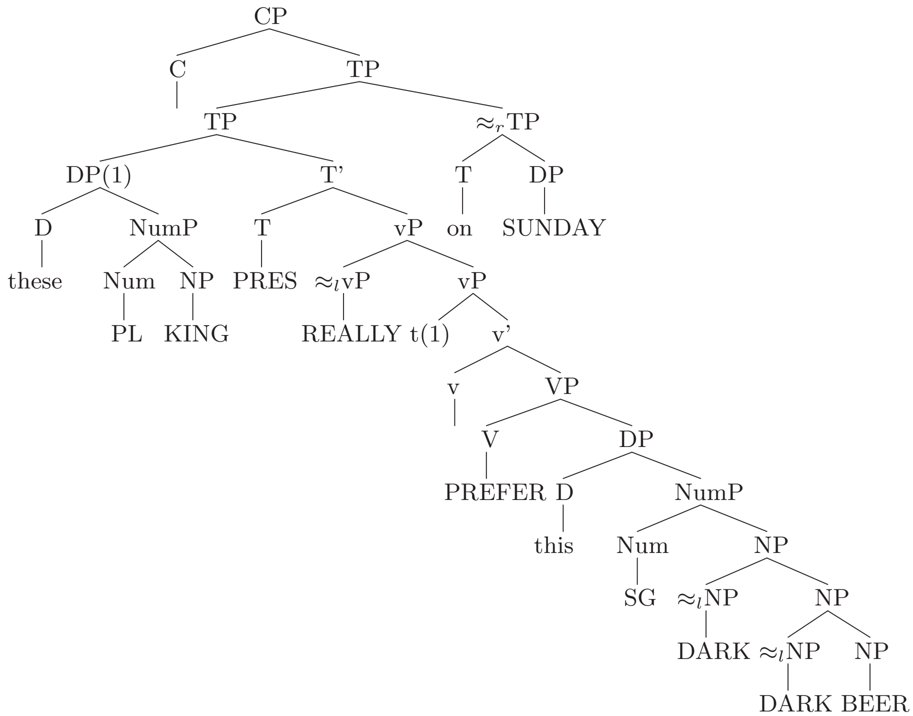




::=T C
on::=D $\approx_r$T
SUNDAY::D
o
PRES::=v +D T
REALLY::$\approx_l$v
::=V =D v
these::=Num D
PREFER::=D V
PL::=N Num
KING::N
this::=Num D
SG::=N Num
DARK::$\approx_l$N
DARK::$\approx_l$N
BEER::N



C
=T C
T
$\approx_r$T
T
=D $\approx_r$T
D
+D T,-D
on
SUNDAY
=v +D T
v,-D
PRES
$\approx_l$v
v,-D
REALLY
=D v
D
=V =D v
V
=Num D
Num
=D V
D
these
=N Num
N
PREFER
=Num D
Num
PL
KING
this
=N Num
N
SG
$\approx_l$N
N
DARK
$\approx_l$N
N
DARK
BEER


CP
C
TP
TP
≈r TP
DP(1)
T'
T
DP
D
NumP
T
vP
on
SUNDAY
these
Num
NP
PRES
≈l vP
vP
PL
KING
REALLY
t(1)
v'
v
VP
V
DP
PREFER
D
NumP
this
Num
NP
SG
≈l NP
NP
DARK
≈l NP
NP
DARK
BEER

8.3 parsing with persistence and adjunction mgBeamPpa.pl

```
% file: mgBeamPpa.pl.
%   mgBeamPp.pl adds persistent features to the basic parser mgBeamP.pl
%   This file adds adjunction to mgBeamPp.pl
% TODO: Add left,right adjoing *mover*
:- [library(heaps)].

% INITIALIZE AND BEGIN
parse(_/LexTs,Input,D) :- % last arg is derivation!
    startCategory(F),
    memberOnce(cat(F)/Ts,LexTs),
    singleton_heap(Queue,[],(cat(F)/Ts,[cat(F)],[],[],[],[],DF)), % last arg is derivation!
    singleton_heap(Beam,-1,(Input,Queue,DF)),  % last arg is derivation!
%     portray_beam(Beam), % for tracing only
    extendBeam(LexTs,Beam,D).

% EXTEND THE BEAM RECURSIVELY
extendBeam(LexTs,Beam0,D) :-
    get_from_heap(Beam0,P0,(In,Q0,A0),Beam1), % pop most probable parse
    ( success(In,Q0), D=A0
    ; get_from_heap(Q0,_,(R/Ts,Anc,TI,Movers,Ancs,MIs,A),Q), % pop leftmost cat
      findall(Parse,(member(T,Ts),infer(T,Anc,TI,Movers,Ancs,MIs,A,LexTs,(In,Q,A0),Parse)
                    ;R=cat(F),adjoin(F,Ts,Anc,TI,Movers,Ancs,MIs,A,LexTs,(In,Q,A0),Parse)
                    ),New),
      length(New,NumberOfOptions),
      ( NumberOfOptions>0,
        P is (1/NumberOfOptions)*P0, % uniform probability over next steps
        P < -0.00001 -> % Simple pruning rule: improbability bound (cf Roark'01)
        insertAll(New,P,Beam1,Beam),
        %portray_beam(Beam),        % for tracing only
        extendBeam(LexTs,Beam,D)
      ; extendBeam(LexTs,Beam1,D)
      )
    ; empty_heap(Q0),  %portray_beam(Beam1), % for tracing only
      extendBeam(LexTs,Beam1,D)
    ).

% STEPS:  adjoin(F,Ts,Anc,I,Movers,MIs,Derivation,Lex,(Input0,Queue0),(Input,Queue))
adjoin(F,FTs,Anc,TI,Ms0,Ancs0,MIs0,x/[B,C],LexTs,(In,Q0,A),(In,Q,A)) :-  % UNADJOIN
    append01(TI,TI0,TI1), % extend tree index TI with 0 and 1
    (  memberOnce(ra(F)/CTs,LexTs),
       add_to_heap(Q0,TI0,(cat(F)/FTs,Anc,TI0,Ms0,Ancs0,MIs0,C),Q1),
       least(TI1,MIs0,Least),
       add_to_heap(Q1,Least,(ra(F)/CTs,[ra(F)],TI1,[],[],[],B),Q)
    ;  memberOnce(la(F)/CTs,LexTs),
       least(TI1,MIs0,Least),
       add_to_heap(Q0,Least,(cat(F)/FTs,Anc,TI1,Ms0,Ancs0,MIs0,C),Q1),
       add_to_heap(Q1,TI0,(la(F)/CTs,[la(F)],TI0,[],[],[],B),Q)
    ).

% STEPS:  infer(T,I,Movers,MIs,Derivation,Lex,(Input0,Queue0),(Input,Queue))
infer(Words/[],Anc,_TI,Ms,Ancs,_MIs,(Words:Anc)/[],_Lex,(In0,Q,A),(In,Q,A)) :- % SCAN
    Ms=[], Ancs=[], append(Words,In,In0). %, format('~w~n',[scan:Words]).

infer(sel(F)/[FT|FTs],Anc,TI,Ms0,Ancs0,MIs0,Root/[B,C],LexTs,(In,Q0,A),(In,Q,A)) :-  % UNMERGE
```

```
    terminal([FT|FTs],Terminals,NonTerminals),
    append01(TI,TI0,TI1), % extend tree index TI with 0 and 1
    (  Terminals=[_|_],  % unmerge1: unmerge comp
       \+ member(neg(F)/_,Ms0),     % shortest move constraint EPP
       memberOnce(cat(F)/CTs,LexTs), Root=x,
       add_to_heap(Q0,TI0,(sel(F)/Terminals,[sel(F)|Anc],TI0,[],[],[],B),Q1),
       least(TI1,MIs0,Least),
       add_to_heap(Q1,Least,(cat(F)/CTs,[cat(F)],TI1,Ms0,Ancs0,MIs0,C),Q)
    ;  NonTerminals=[_|_], % unmerge2: unmerge spec
       \+ member(neg(F)/_,Ms0),     % shortest move constraint EPP
       memberOnce(cat(F)/CTs,LexTs), Root=x,
       least(TI1,MIs0,Least),
       add_to_heap(Q0,Least,(sel(F)/NonTerminals,[sel(F)|Anc],TI1,Ms0,Ancs0,MIs0,B),Q1),
       add_to_heap(Q1,TI0,(cat(F)/CTs,[cat(F)],TI0,[],[],[],C),Q)
    ;  Terminals=[_|_],  % unmerge3: unmerge a comp mover
       selectMAI(cat(F)/CTs,OtherA,OtherI,Ms0,Ms,Ancs0,Ancs,MIs0,MIs), Root=x,
       add_to_heap(Q0,TI,(sel(F)/Terminals,[sel(F)|Anc],TI,[],[],[],B),Q1),
       least(OtherI,MIs,Least),
       add_to_heap(Q1,Least,(cat(F)/CTs,[cat(F)|OtherA],OtherI,Ms,Ancs,MIs,C),Q)
    ;  NonTerminals=[_|_], % unmerge4: unmerge a spec mover
       selectMAI(cat(F)/CTs,OtherA,OtherI,Ms0,Ms,Ancs0,Ancs,MIs0,MIs), Root=x,
       least(TI,MIs,Least),
       add_to_heap(Q0,Least,(sel(F)/NonTerminals,[sel(F)|Anc],TI,Ms,Ancs,MIs,B),Q1),
       add_to_heap(Q1,OtherI,(cat(F)/CTs,[cat(F)|OtherA],OtherI,[],[],[],C),Q)
    ;  Terminals=[_|_], % unmerge3: unmerge a comp mover EPP case
       selectMAI(eppcat(F)/CTs,OtherA,OtherI,Ms0,Ms,Ancs0,Ancs,MIs0,MIs), Root=xx,
       add_to_heap(Q0,TI,(sel(F)/Terminals,[sel(F)|Anc],TI,[],[],[],B),Q1),
       least(OtherI,MIs,Least),
       add_to_heap(Q1,Least,(cat(F)/CTs,[cat(F)|OtherA],OtherI,Ms,Ancs,MIs,C),Q)
    ;  NonTerminals=[_|_], % unmerge4: unmerge a spec mover EPP case
       selectMAI(eppcat(F)/CTs,OtherA,OtherI,Ms0,Ms,Ancs0,Ancs,MIs0,MIs), Root=xx,
       least(TI,MIs,Least),
       add_to_heap(Q0,Least,(sel(F)/NonTerminals,[sel(F)|Anc],TI,Ms,Ancs,MIs,B),Q1),
       add_to_heap(Q1,OtherI,(cat(F)/CTs,[cat(F)|OtherA],OtherI,[],[],[],C),Q)
    ).

infer(pos(F)/[FT|FTs],Anc,TI,Ms0,Ancs0,MIs0,o/[B],LexTs,(In,Q0,A),(In,Q,A)) :- % UNMOVE
    \+ member(neg(F)/_,Ms0),     % shortest move constraint
    ( memberOnce(neg(F)/NTs,LexTs), % unmove1
      append01(TI,TI0,TI1),
      least(TI1,[TI0|MIs0],Least),
      add_to_heap(Q0,Least,
       (pos(F)/[FT|FTs],[pos(F)|Anc],TI1,[neg(F)/NTs|Ms0],[[neg(F)]|Ancs0],[TI0|MIs0],B),
                  Q)
    ; selectMAI(neg(F)/NTs,OtherA,OtherI,Ms0,Ms,Ancs0,Ancs,MIs0,MIs), % unmove2
      least(TI,[OtherI|MIs],Least),
      add_to_heap(Q0,Least,
       (pos(F)/[FT|FTs],[pos(F)|Anc],TI,[neg(F)/NTs|Ms],[[neg(F)|OtherA]|Ancs],[OtherI|MIs],B),
                  Q)
    ; memberOnce(neg(F)/NTs,LexTs), % unmove1 SC : we push a neg(F) back into movers
      append01(TI,TI0,TI1),
      least(TI1,[TI0|MIs0],Least),
      add_to_heap(Q0,Least,
       (pos(F)/[FT|FTs],[pos(F)|Anc],TI1,[neg(F)/[neg(F)/NTs]|Ms0],[[neg(F)]|Ancs0],[TI0|MIs0],B),
                  Q)
    ; selectMAI(neg(F)/NTs,OtherA,OtherI,Ms0,Ms,Ancs0,Ancs,MIs0,MIs), % unmove2 SC
```

```
      least(TI,[OtherI|MIs],Least),
      add_to_heap(Q0,Least,
       (pos(F)/[FT|FTs],[pos(F)|Anc],TI,[neg(F)/[neg(F)/NTs]|Ms],[[neg(F)|OtherA]|Ancs],[OtherI|MIs],B),
                 Q)
    ; epp(F),  % EPP variant of unmove1: insert mover
      memberOnce(cat(F)/NTs,LexTs),
      append01(TI,TI0,TI1),
      least(TI1,[TI0|MIs0],Least),
      add_to_heap(Q0,Least,
       (pos(F)/[FT|FTs],[pos(F)|Anc],TI1,[neg(F)/[eppcat(F)/NTs]|Ms0],[[epp(F)]|Ancs0],[TI0|MIs0],B),
                 Q)
    ).

% DEFINITION OF SUCCESS: THE INPUT IS EMPTY, THE PARSE QUEUE IS EMPTY
success([],Q) :- empty_heap(Q).

% terminal(Cats,Terminals,Nontermials) split Cats into Terminals/Nonterminals
terminal([],[],[]).
terminal([Ws/[]|Ts],[Ws/[]|Trms],NonTrms) :- !, terminal(Ts,Trms,NonTrms).
terminal([T|Ts],Trms,[T|NonTrms]) :- terminal(Ts,Trms,NonTrms).

memberOnce(E,[E|_]) :- !.
memberOnce(E,[_|L]) :- memberOnce(E,L).

% insertAll(Es,P,B0,B) B is result of adding all Es to B0 with priority P
insertAll([],_,B,B).
insertAll([E|Es],P,B0,B) :- add_to_heap(B0,P,E,B1), insertAll(Es,P,B1,B).

% append(L,L0,L1) L0 is L with 0 appended, L1 has 1 appended
append01([],[0],[1]).
append01([E|L],[E|L0],[E|L1]) :- append01(L,L0,L1).

% select mover, ancestors and index (note that mover is embedded!)
selectMAI(E,A,I,[_/Ts|Es],Es,[A|As],As,[I|Is],Is) :- member(E,Ts).
selectMAI(E,A,I,[F|Fs],[F|Gs],[B|Cs],[B|Ds],[J|Is],[J|Js]) :- selectMAI(E,A,I,Fs,Gs,Cs,Ds,Is,Js).

portray_beam(B) :- heap_to_list(B,L), heap_size(B,S),
    format(’~n~w~w~n’,[S,’ parses in beam:’]), portray_parseN(L,1).

showRootsOnly([],[]). % only roots to make trace more readable
showRootsOnly([-(P,(T/_,_Anc,TI,Ms0,_Ancs,MsI,A))|L],[-(P,(T,TI,Ms,MsI,A))|RL]) :-
        rootsOnly(Ms0,Ms),
        showRootsOnly(L,RL).

rootsOnly([],[]).   rootsOnly([R/_|Ts],[R|Rs]) :- rootsOnly(Ts,Rs).

portray_parseN([],_). % portray each parse, numbering them from 1
portray_parseN([-(P,(In,Q,A))|Items],N) :-
    heap_size(Q,S), heap_to_list(Q,QL0), showRootsOnly(QL0,QL),
    format(’~w~w~w~w~w~w~w~w~w~w~w~w~w~n’,[N,’(’,S,’). ’,’(’,P,’,(’,In,’,’,QL,’,’,A,’))’]),
    N1 is N+1,
    portray_parseN(Items,N1).

least(I,[],I).  % least(I,Is,J) = J is the least index among I and Is
least(I,[J|Js],Least) :- J@<I -> least(J,Js,Least); least(I,Js,Least).
```

```
% Tree drawing tools, lexical tree builder, examples
:- [pp_tree,wish_tree,lexBuild,beautify,dt2bt,dt2stt,dt2xb].
%:- [mg0t0].
%:- lexBuild(LexT), parse(LexT,[these,'PL','KING','PRES','PREFER',this,'SG','BEER'],T),
:- [mg0t1].
%:- lexBuild(L), wish_tree(L).
%:- lexBuild(LexT), parse(LexT,['JOHN','KNOW',this,'SG','KING','PRES','LAUGH'],T),
%:- lexBuild(LexT), parse(LexT,['JOHN','PRES','KNOW',this,'SG','KING','PRES','LAUGH'],T),
%:- lexBuild(LexT), parse(LexT,[these,'PL','KING','PRES','PREFER',this,'SG','BEER'],T),
%:- lexBuild(LexT), parse(LexT,['MARY','PRES','KNOW','JOHN','PRES','PREFER',this,'SG','BEER'],T),
%:- lexBuild(LexT), parse(LexT,['MARY','PRES','KNOW',which,'SG','BEER','JOHN','PRES','PREFER'],T),
%:- lexBuild(LexT), parse(LexT,['MARY','PRES','KNOW',which,'SG','DARK','BEER','JOHN','PRES','PREFER'],T),
%:- lexBuild(LexT), parse(LexT,[which,'SG','DARK','BEER','JOHN','PRES','PREFER',on,'SUNDAY'],T),
%:- lexBuild(LexT), parse(LexT,['JOHN','PRES','KNOW',these,'PL','KING','PRES','PREFER',
%                               this,'SG','BEER'],T),
%:- lexBuild(LexT), parse(LexT,['JOHN','PRES','KNOW',which,'SG','BEER',these,'PL','KING',
%                               'PRES','PREFER'],T),
%:- lexBuild(LexT), parse(LexT,[these,'PL','KING','PRES','PREFER',this,'SG','BEER',tomorrow],T),
%:- lexBuild(LexT), parse(LexT,[these,'PL','KING','PRES','REALLY','PREFER',this,'SG','BEER'],T),
%:- lexBuild(LexT), parse(LexT,[these,'PL','KING','PRES','REALLY','PREFER',this,'SG',
%                               'DARK','DARK','BEER'],T),
%:- lexBuild(LexT), parse(LexT,[these,'PL','KING','PRES','REALLY','PREFER',this,'SG',
%                               'DARK','DARK','BEER',tomorrow],T),
%:- lexBuild(LexT), parse(LexT,[these,'PL','KING','PRES','REALLY','PREFER',this,'SG',
%                               'DARK','DARK','BEER',on,'SUNDAY'],T),
% Out of global stack on this next one:
%       optimize probability distribution? bigger machine? optimize code? Yes!
:- lexBuild(LexT), parse(LexT,['JOHN','PRES','KNOW',which,'SG','DARK','DARK','BEER',these,
                               'PL','KING','PRES','REALLY','PREFER',on,'SUNDAY'],T),
       btfy(T,D), wish_tree(D), sleep(3),
       dt2bt(T,BT), btfy_bt(BT,BBT), wish_tree(BBT), sleep(3),
       dt2stt(T,STT), btfy_stt(STT,BSTT), wish_tree(BSTT), sleep(3),
       dt2xb(T,XB), btfy_xb(XB,BXB), wish_tree(BXB).
```

References

[1] Abels, K., and Neeleman, A. Linear asymmetries and the LCA. *Syntax 12*, 1 (2012), 25–74.

[2] Frey, W., and Gärtner, H.-M. On the treatment of scrambling and adjunction in minimalist grammars. In *Proceedings, Formal Grammar'02* (Trento, 2002).

[3] Graf, T. Closure properties of minimalist derivation tree languages. In *Logical Aspects of Computational Linguistics, LACL'11* (2011).

[4] Hale, J., and Stabler, E. P. Strict deterministic aspects of minimalist grammars. In *Logical Aspects of Computational Linguistics, LACL'05*, Lecture Notes in Artificial Intelligence LNCS-3492. Springer, NY, 2005, pp. 162–176.

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

[6] Hunter, T. *Relating Movement and Adjunction in Syntax and Semantics*. PhD thesis, University of Maryland, 2010.

[7] Kayne, R. S. *The Antisymmetry of Syntax*. MIT Press, Cambridge, Massachusetts, 1994.

[8] Kayne, R. S. A note on prepositions, complementizers, and word order universals. In *Parameters and Universals*. Oxford University Press, Oxford, 2000.

[9] Kayne, R. S. Antisymmetry and Japanese. *English Linguistics 20* (2003), 1–40. Reprinted in R. S. Kayne, *Movement and Silence*, NY: Oxford University Press, 2005.

[10] Kayne, R. S. Prepositions as probes. In *Structures and Beyond: The Cartography of Syntactic Structures Volume 3*, A. Belletti, Ed. Oxford University Press, Oxford, 2004.

[11] Kayne, R. S. Antisymmetry and the lexicon. Manuscript, http://ling.auf.net/lingbuzz, 2007.

[12] Kobele, G. M. On late adjunction in minimalist grammars. Workshop on Multiple Context-Free Grammars and Related Formalisms, MCFG+, National Institute of Informatics, Tokyo, 2010.

[13] Kobele, G. M. Minimalist tree languages are closed under intersection with recognizable tree languages. In *Logical Aspects of Computational Linguistics, LACL'11* (2011).

[14] Kobele, G. M., and Michaelis, J. Locality, late adjunction and extraposition. Presented at the European Summer School for Logic Language and Information, ESSLLI'09, 2009.

[15] Kobele, G. M., Retoré, C., and Salvati, S. An automata-theoretic approach to minimalism. In *Model Theoretic Syntax at 10. ESSLLI'07 Workshop Proceedings* (2007), J. Rogers and S. Kepser, Eds.

[16] Koopman, H. Korean (and Japanese) morphology from a syntactic perspective. *Linguistic Inquiry 36*, 4 (2005), 601–633.

[17] Koopman, H. On the parallelism of DPs and clauses in Kisongo Maasai. In *Verb First*, A. Carnie, S. Dooley, and H. Harley, Eds. John Benjamins, Philadelphia, 2005.

[18] Koopman, H. Agreement configurations: In defense of spec-head. In *Agreement Systems*, C. Boeckx, Ed. John Benjamins, Philadelphia, 2006, pp. 159–200.

[19] Koopman, H., Sportiche, D., and Stabler, E. *An Introduction to Syntactic Analysis and Theory*. UCLA Lecture Notes, forthcoming, 2012.

[20] Koopman, H., and Szabolcsi, A. *Verbal Complexes*. MIT Press, Cambridge, Massachusetts, 2000.

[21] Mahajan, A. Eliminating head movement. In *The 23rd Generative Linguistics in the Old World Colloquium, GLOW '2000, Newsletter #44* (2000), pp. 44–45.

[22] Mainguy, T. A probabilistic top-down parser for minimalist grammars. http://arxiv.org/abs/1010.1826v1, 2010.

[23] Michaelis, J. *On Formal Properties of Minimalist Grammars*. PhD thesis, Universität Potsdam, 2001. *Linguistics in Potsdam 13*, Universitätsbibliothek, Potsdam, Germany.

[24] Michaelis, J., Mönnich, U., and Morawietz, F. On minimalist attribute grammars and macro tree transducers. In *Linguistic Form and its Computation*, C. Rohrer, A. Rossdeutscher, and H. Kamp, Eds. CSLI Publications, Stanford, California, 2001, pp. 287–326.

[25] Mönnich, U. Minimalist syntax, multiple regular tree grammars, and direction preserving tree transductions. In *Model Theoretic Syntax at 10. ESSLLI'07 Workshop Proceedings* (2007), J. Rogers and S. Kepser, Eds.

[26] Morawietz, F. *Two Step Approaches to Natural Language Formalisms*. PhD thesis, University of Tübingen, 2001.

[27] Roark, B. Probabilistic top-down parsing and language modeling. *Computational Linguistics 27*, 2 (2001), 249–276.

[28] Roark, B. Robust garden path parsing. *Natural Language Engineering 10*, 1 (2004), 1–24.

[29] Roark, B., and Johnson, M. Efficient probabilistic top-down and left-corner parsing. In *Proceedings of the 37th Annual Meeting of the Association for Computational Linguistics* (1999), pp. 421–428.

[30] Sportiche, D. *Partitions and Atoms of Clause Structure : Subjects, Agreement, Case and Clitics*. Routledge, NY, 1998.

[31] Sportiche, D. Reconstruction, binding, and scope. In *The Blackwell Companion to Syntax, Volume IV*, M. Everaert and H. van Riemsdijk, Eds. Blackwell, Oxford, 2006.

[32] Stabler, E. P. Recognizing head movement. In *Logical Aspects of Computational Linguistics*, P. de Groote, G. Morrill, and C. Retoré, Eds., Lecture Notes in Artificial Intelligence, No. 2099. Springer, NY, 2001, pp. 254–260.

[33] Stabler, E. P. Comparing 3 perspectives on head movement. In *From Head Movement and Syntactic Theory, UCLA/Potsdam Working Papers in Linguistics*, A. Mahajan, Ed. UCLA, 2003, pp. 178–198. Available at http://www.humnet.ucla.edu/humnet/linguistics/people/stabler.

[34] Stabler, E. P. Computational perspectives on minimalism. In *Oxford Handbook of Linguistic Minimalism*, C. Boeckx, Ed. Oxford University Press, Oxford, 2011, pp. 617–641.

[35] Stabler, E. P. Top-down recognizers for MCFGs and MGs. In *Proceedings of the Workshop on Cognitive Modeling and Computational Linguistics (CMCL), 49th Annual Meeting of the Association for Computational Linguistics* (2011), F. Keller and D. Reitter, Eds.

[36] Stabler, E. P. Bayesian, minimalist, incremental syntactic analysis. *Topics in Cognitive Science forthcoming* (2012).

[37] Villemonte de la Clergerie, E. Parsing MCS languages with thread automata. In *Proceedings, 6th International Workshop on Tree Adjoining Grammars and Related Frameworks, TAG+6* (2002).

A basic, bottom up, MG[+SMC] rules

This appendix just restates the starting point for the extensions considered above. For $s, t \in \Sigma^*$, $\cdot \in \{:, ::\},$, $\gamma \in F^*$, $\delta \in F^+$, for multisets of chains ('movers') μ, μ_0, μ_1

$$\frac{s :: \texttt{=}f\gamma, \emptyset \qquad t \cdot f, \mu}{st : \gamma, \mu} \ (\bullet_1) \text{ lexical item selects a non-mover}$$

$$\frac{s : \texttt{=}f\gamma, \mu_0 \qquad t \cdot f, \mu_1}{ts : \gamma, \mu_0 \uplus \mu_1} \ (\bullet_2) \text{ derived item selects a non-mover}$$

$$\frac{s \cdot \texttt{=}f\gamma, \mu_0 \qquad t \cdot f\delta, \mu_1}{s : \gamma, \mu_0 \uplus \mu_1 \uplus \{t : \delta\}} \ (\bullet_3) \text{ any item selects a mover}$$

$$\frac{s : \texttt{+}f\gamma, \mu \uplus \{t : \texttt{-}f\}}{ts : \gamma, \mu} \ (\circ_1) \text{ final move of licensee}$$

$$\frac{s : \texttt{+}f\gamma, \mu \uplus \{t : \texttt{-}f\delta\}}{s : \gamma, \mu \uplus \{t : \delta\}} \ (\circ_2) \text{ nonfinal move of licensee.}$$

Here $\uplus$ is multiset union and SMC is enforced at every step.

B top-down MG[+SMC,+SpIC$_{mrg}$] rules

TD parser rules from [36]: the rules above are flipped over to be TD, with +SpIC$_{mrg}$.

$$\frac{}{input,\ (\mathrm{C}(x), \emptyset)_\epsilon} \ (\textsc{start})\ \ell[\mathrm{C}(x)], \text{ for start category C}$$

$$\frac{w * input,\ (t[w], \emptyset)_i * q}{input,\ q} \ (\textsc{scan})$$

$$\frac{input,\ (t[\texttt{=}f(x)], \mu)_i * q}{input,\ (\texttt{=}f(\Sigma x), \emptyset)_{i0} * (f(y), \mu)_{i1} * q} \ (\bullet_1)\ \ell[f(y)] \wedge \Sigma x \neq \epsilon$$

$$\frac{input,\ (t[\texttt{=}f(x)], \mu)_i * q}{input,\ (\texttt{=}f(\overline{\Sigma}x), \mu)_{i1} * (f(y), \emptyset)_{i0} * q} \ (\bullet_2)\ \ell[f(y)] \wedge \overline{\Sigma}x \neq \epsilon$$

$$\frac{input,\ (t[\texttt{=}f(x)],\ u[f(y)]_j \uplus \mu)_i * q}{input,\ (\texttt{=}f(\Sigma x), \emptyset)_i * (f(y), \mu)_j * q} \ (\bullet_3)\ \Sigma x \neq \epsilon$$

$$\frac{input,\ (t[\texttt{=}f(x)],\ u[f(y)]_j \uplus \mu)_i * q}{input,\ (\texttt{=}f(\overline{\Sigma}x), \mu)_i * (f(y), \emptyset)_j * q} \ (\bullet_4)\ \overline{\Sigma}x \neq \epsilon$$

$$\frac{input,\ (t[\texttt{+}f(x)], \mu)_i * q}{input,\ (\texttt{+}f(x), \texttt{-}f(y)_{i0} \uplus \mu)_{i1} * q} \ (\circ_1)\ \ell[\texttt{-}f(y)]$$

$$\frac{input,\ (t[\texttt{+}f(x)],\ u[\texttt{-}f(y)]_j * \mu)_i * q}{input,\ (\texttt{+}f(x), \texttt{-}f(y)_j \uplus \mu)_i * q} \ (\circ_2)$$