Modular Minimalist Grammar

Edward Stabler UCLA

2024-09-02

- Modular minimalist grammar composed over workspaces
- Interfaces
- Language model composed from grammar and interfaces
- Language model as 1 transduction (modularity breached!)

These slides, code, and (soon) draft paper: https://github.com/epstabler/mgt.
The draft paper has the details for references mentioned in these slides/comments

Thanks to audiences at the MG+1, MG+2 meetings for helpful suggestions.

2024-09-02

Modular Minimalist Grammar
Edward Stabler UCLA
2024-09-02

MG derivational steps are broken up then reassembled,

The theory becomes **modular** in the sense that the syntax (the syntactic objects, movement relations, case and agreement properties, etc.) emerges from the interaction of separately defined processes. There are 'breaches' in modularity, where a 'later' process influences an 'earlier' one, but these breaches are restricted.

This modular approach makes MGs *easier to change*.
For many changes you might want to make, only one or two components will need to be modified, leaving the rest intact.

The last step, putting everything into 1 transduction, is reminiscent of Collins&Stabler and some other Chomskian, minimalist proposals, and raises similar issues. I think the scope and signficance of those issues is perhaps clearer here.

(De)composing each derivational step

5 grammatical functions: mrg, t, smc, match, ck
+ 3 bureaucratic functions:

the 'cons' function :
$1 : [2,3] = [1,2,3]$

the 'tail' function :
$\text{tail}\ [1,2,3] = [2,3]$

the 'pair concatenation' function:
$([1,2],[a,b])$ +++ $([3],[c]) = ([1,2,3],[a,b,c])$

2024-09-02

└(De)composing each derivational step

(De)composing each derivational step
5 grammatical functions: mrg, t, smc, match, ck
+ 3 bureaucratic functions:
the 'cons' function : 1 : [2,3] = [1,2,3]
the 'tail' function : tail [1,2,3] = [2,3]
the 'pair concatenation' function: ([1,2],[a,b]) +++ ([3],[c]) = ([1,2,3],[a,b,c])

One MG derivational step will be composed from 8 simple functions.
We cover all the subcases of earlier MG, but without breaking the rule up that way.

The 3 bureaucratic functions are very simple.

Setup

Grammar: a finite set of lexical items, (ph form, label) pairs:
g = { (jo, D), (which, N−∘D.Wh), (likes, D.D−∘V) }

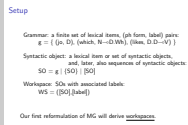
Syntactic object: a lexical item or set of syntactic objects,
and, later, also sequences of syntactic objects:
SO = g | {SO} | [SO]

Workspace: SOs with associated labels:
WS = ([SO],[label])

Our first reformulation of MG will derive workspaces.

2024-09-02

└Setup



We will define 5 linguistic functions over these types of things.

Initially, the SOs are lexical items or sets built from those (indicated by curly braces), but later we allow also sequences built from lexical items (indicated by square brackets) and slight alterations in the phonetic contents of the lexical items.

A workspace is a pair – a sequence of SOs paired with the sequence of labels of those SOs.

(As will become clear, WSs are basically similar to the derived structures of earlier MGs.)

5 linguistic functions

$$(\text{match})\ \frac{\mathsf{WS}_1, \ldots, \mathsf{WS}_i \quad \text{initially, for } i = 2}{\text{WS of matching elements, WS' non-matching elements}}$$

$$(\text{mrg})\ \frac{\mathsf{SO}_1, \ldots, \mathsf{SO}_n}{\{\mathsf{SO}_1, \ldots, \mathsf{SO}_n\}}$$

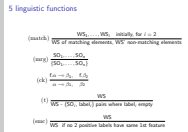
$$(\text{ck})\ \frac{\mathsf{f}.\alpha \multimap \beta_1, \quad \mathsf{f}.\beta_2}{\alpha \multimap \beta_1, \quad \beta_2}$$

$$(\text{t})\ \frac{\text{WS}}{\text{WS - } (\mathsf{SO}_i, \mathsf{label}_i) \text{ pairs where label}_i \text{ empty}}$$

$$(\text{smc})\ \frac{\text{WS}}{\text{WS} \quad \text{if no 2 positive labels have same 1st feature}}$$

2024-09-02

└5 linguistic functions



Linearization is left to PF interface. . .

match finds neg element of WS_1. If WS_1 has pos match SO_j,label_j then
(move-over-merge) SO_j = head SO of WS_2, and use label_j
(else) head WS_2 is a pos match.
Other non-matching elements partitioned into separate workspace WS'

Function ck is *modus ponens*, the *law of detachment*

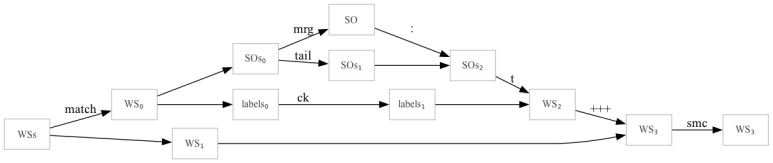
Function t removes the inert, 'trivial' SOs, those with no features. . .
An example will make this clear, below

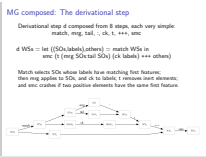
MG composed: The derivational step

Derivational step d composed from 8 steps, each very simple:
match, mrg, tail, :, ck, t, +++, smc

d WSs = let ((SOs,labels),others) = match WSs in
smc (t (mrg SOs:tail SOs) (ck labels) +++ others)

Match selects SOs whose labels have matching first features;
then mrg applies to SOs, and ck to labels; t removes inert elements;
and smc crashes if two positive elements have the same first feature.





⇐ This is the actual Haskell definition in my github implementation.

The python implementation is not quite so elegant, but similar:

```
def d(wss):
    (matches, others) = match(wss)
    return smc( t( WS( [mrg(matches._sos)] + matches._sos[1:], ck(matches._labels) ) ).pappend(others) )
```

To avoid the 'let...' construction, we could use projection functions to get the components from the output of match.

'Let...' constructions enhance readability, and in programming languages they can enhance efficiency by defining 'staged computations' that avoid recalculation of results.

Cf https://en.wikipedia.org/wiki/Let_expression, Davies&Pfenning'01

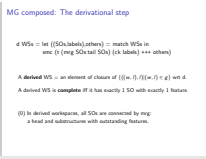
MG composed: The derivational step

d WSs = let ((SOs,labels),others) = match WSs in
smc (t (mrg SOs:tail SOs) (ck labels) +++ others)

A **derived** WS = an element of closure of $\{((w, l), l) | (w, l) \in g\}$ wrt d.

A derived WS is **complete** iff it has exactly 1 SO with exactly 1 feature.

(0) In derived workspaces, all SOs are connected by mrg:
a head and substructures with outstanding features.

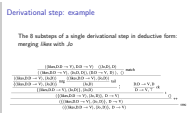


(0) means that we can view the workspace as a moving, labeled 'window' on the head SO, a kind of 'locus of attention'

Derivational step: example

The 8 substeps of a single derivational step in deductive form: merging *likes* with *Jo*

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{((\text{likes},\text{D} \multimap \text{V}),\text{D} \multimap \text{V})\quad((\text{Jo},\text{D}),\text{D})}{(\,(\text{likes},\text{D} \multimap \text{V}),\ (\text{Jo},\text{D}),\text{D})),\ (\text{D} \multimap \text{V},\ \text{D})\,),\ ()}\ \text{match}}{((\text{likes},\text{D} \multimap \text{V}),(\text{Jo},\text{D}))}\ \text{mrg}\ \dfrac{((\text{likes},\text{D} \multimap \text{V}),(\text{Jo},\text{D}))}{(\text{Jo},\text{D})}\ \text{tail}}{\{(\text{likes},\text{D} \multimap \text{V}),(\text{Jo},\text{D})\}}\ :\ \dfrac{\text{D.D} \multimap \text{V},\text{D}}{\text{D} \multimap \text{V},\top}\ \text{ck}}{\{(\text{likes},\text{D} \multimap \text{V}),(\text{Jo},\text{D})\},(\text{Jo},\text{D})}\ \text{t}}{(\{(\text{likes},\text{D} \multimap \text{V}),(\text{Jo},\text{D})\},\ \text{D} \multimap \text{V})}\ ()}{(\{(\text{likes},\text{D} \multimap \text{V}),(\text{Jo},\text{D})\},\ \text{D} \multimap \text{V})}\ ++}{(\{(\text{likes},\text{D} \multimap \text{V}),(\text{Jo},\text{D})\},\ \text{D} \multimap \text{V})}\ \text{smc}$$



Instead of merging a verb with a direct object in 1 specialized step, we do it in 8 simpler steps that cover all the cases.
A note about the notation for labels:

(praise, D.D⊸V)	Given 2 DPs, a VP.
	Dot for conjunction.
	Empty conjunction written $\top$, for top, true.
	Antecedent features are 'negative'.
(which, N⊸D.Wh)	Given NP, a wh DP.
(student, $\top$ ⊸ N)	A noun(phrase).
	Usually written: (student, N).
$\top$	A label with no features.

In the derivation to the left here, when D is checked, $\top$ remains. So that element is inert, and is 'forgotten' from the workspace by rule t.

Derivational step: example

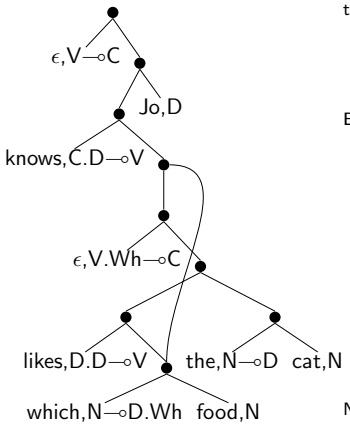
The 8 substeps of a single derivational step in deductive form: merging *likes* with *who*

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{((\text{likes},\text{D} \multimap \text{V}),\text{D} \multimap \text{V})\quad((\text{who},\text{D.Wh}),\text{D.Wh})}{(\,(\text{likes},\text{D} \multimap \text{V}),\ (\text{who},\text{D.Wh})),\ (\text{D.D} \multimap \text{V},\ \text{D.Wh})\,),\ ()}\ \text{match}}{((\text{likes},\text{D} \multimap \text{V}),(\text{who},\text{D.Wh}))}\ \text{mrg}\ \dfrac{((\text{likes},\text{D} \multimap \text{V}),(\text{who},\text{D.Wh}))}{(\text{who},\text{D.Wh})}\ \text{tail}}{\{(\text{likes},\text{D} \multimap \text{V}),(\text{who},\text{D.Wh})\}}\ :\ \dfrac{\text{D.D} \multimap \text{V},\text{D.Wh}}{\text{D} \multimap \text{V},\text{Wh}}\ \text{ck}}{\{(\text{likes},\text{D} \multimap \text{V}),(\text{who},\text{D.Wh})\},(\text{who},\text{D.Wh})}\ \text{t}}{(\{(\text{likes},\text{D} \multimap \text{V}),(\text{who},\text{D.Wh})\},(\text{who},\text{D.Wh})),\ (\text{D} \multimap \text{V},\text{Wh})}\ ()}{(\{(\text{likes},\text{D} \multimap \text{V}),(\text{who},\text{D.Wh})\},(\text{who},\text{D.Wh})),\ (\text{D} \multimap \text{V},\text{Wh})}\ ++}{(\{(\text{likes},\text{D} \multimap \text{V}),(\text{who},\text{D.Wh})\},(\text{who},\text{D.Wh})),\ (\text{D} \multimap \text{V},\text{Wh})}\ \text{smc}$$

The same pattern of rules combines *likes* and *who*, but because *who* has two positive features, the effect of step t is different, and as a result, the resulting workspace has 2 LSOs instead of just 1. This would be accomplished with a different rule in many early MGs, but here it is exactly the same pattern of 8 steps

Since these 8 steps are exactly d, and d is the only rule, we can just show the 8 steps as one, and we do not need to label it.

MG derived structures: Syntactic objects



the ∈ relations for a derived complete SO,

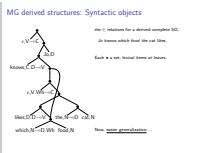
Jo knows which food the cat likes.

Each • a set; lexical items at leaves.

Now, easier generalization. . .

2024-09-02

└MG derived structures: Syntactic objects

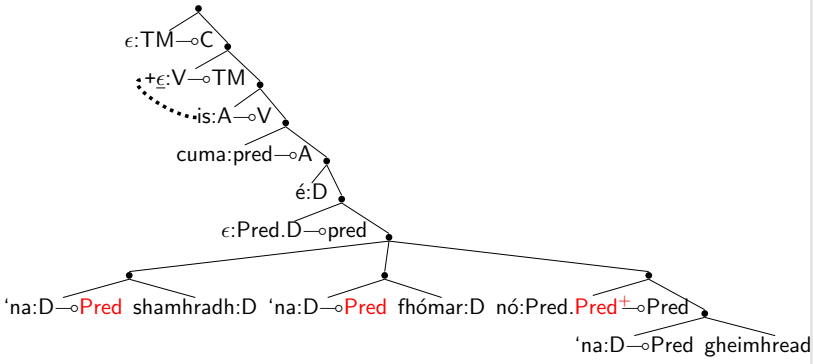


Since the feature checking done by d is so simple, after some practice, I find the SO itself to be the most readable notation for the derivation, though I can understand why linguists prefer highly redundant X-bar-like notations in the literature.

Each set represents 1 application of d to the workspaces corresponding to its children/elements. (This is formalized as transduction ℓ below.)

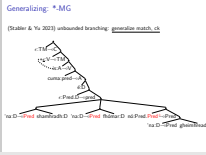
Generalizing: *-MG

(Stabler & Yu 2023) unbounded branching: generalize match, ck



2024-09-02

└Generalizing: *-MG



This could not be done in MG versions before Stabler&Yu, slightly simplfied here

One other detail is discussed in the written version of this talk:

To allow multiple identical elements sisters of head, instead of the usual (global) indexing, merge is also adjusted: it builds *multisets*.

Using multisets is essentially equivalent to indexing, but local.

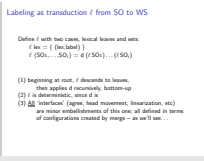
One other detail: What's that dotted line? Head movement.

Treated here not by revising merge, but as a 'post-syntactic' transduction. . .

Labeling as transduction ℓ from SO to WS

Define ℓ with two cases, lexical leaves and sets:
ℓ lex = { (lex,label) }
ℓ {SO_1,...,SO_i} = d (ℓSO_1)...(ℓSO_i)

(1) beginning at root, ℓ descends to leaves,
then applies d recursively, bottom-up
(2) ℓ is deterministic, since d is
(3) All 'interfaces' (agree, head movement, linearization, etc)
are minor embellishments of this one; all defined in terms of configurations created by merge – as we'll see...



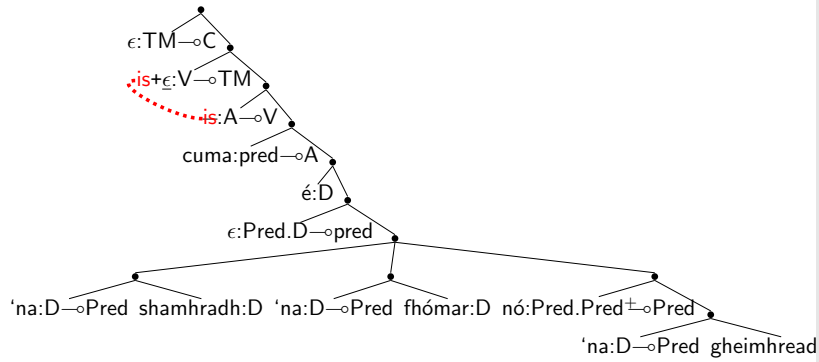
NB: ℓ is a partial function – some merge structures cannot be labeled

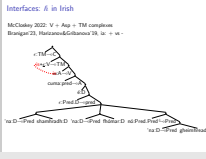
Property (3) is a surprising idea, with analogues in all the major traditions of syntax. As explained in the paper, each syntactic structure has exactly one basic phrase structure 'backbone': All embellishments (case, agree, linearization,...) are then definable as homomorphisms.

Cf. Hornstein's (2024, pp7-8) "Fundamental Principle of Grammar":
(FPG) "All grammatical relations are merge-mediated."
That sounds a lot like (3)! Hornstein suggests that FPG is incompatible with modularity, but that claim uses 'modular' in a sense different from ours (see the comments on slide 1), since he maintains, e.g., that defining merge as set formation and then defining case in terms of a kind spec-head checking *is* compatible with FPG, even though that *is* modular in our sense of the term. But he uses 'modular' more strictly than us... See the paper for more discussion

Interfaces: $\hbar$ in Irish

McCloskey 2022: V + Asp + TM complexes
Branigan'23, Harizanov&Gribanova'19, ia: + vs -

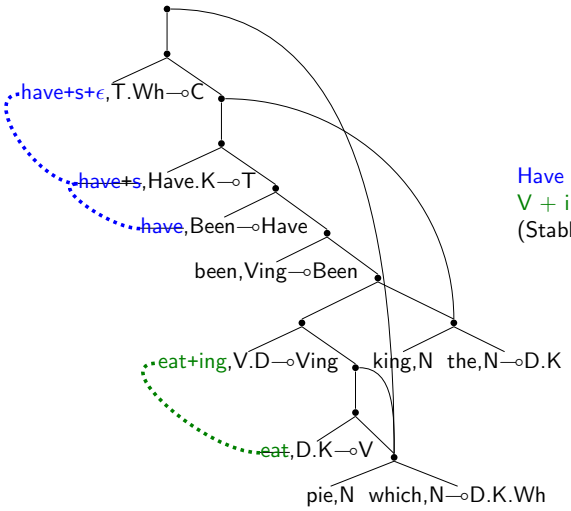




V raises to TM

(McCloskey'22 discusses more complex examples, left for future work)

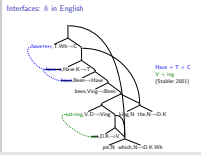
Interfaces: $\hbar$ in English



Have + T + C
V + ing
(Stabler 2001)

2024-09-02

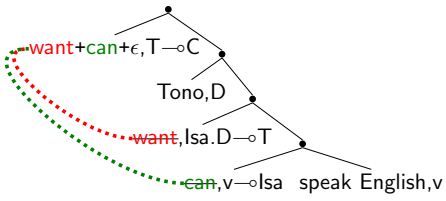
└Interfaces: $\hbar$ in English



The simplified treatment of Eng aux from Stabler 2001, deriving:

which pie have +s +ϵ the king been eat +ing

Interfaces: $\hbar$ in Javanese



Dheen gelem isa ngomong Inggris
he want can speak English

Gelem isa dheen ngomong Inggris?
want can he speak English?

(Cole&al 2008, Branigan 2023) 'tucking in'

2024-09-02

└Interfaces: $\hbar$ in Javanese

In one variety of Javanese, and in some other languages, more than 1 verb can move up to C, 'tucking in' so that linear order corresponds to linear order of the linearized sources.

This could not be done with previous MG head movement mechanisms

Interface: $\hbar$ as transduction

Basic idea: Lexical items marked + must associate with X0 stem
If selector is marked +, move there.

In BU transduction: 'Look ahead' to selector.

Strategy of precise definition:

a. On path from root to leaf, pass "+" to children
b. On way up, at each node:
   if "+": then delete head and pass it (and upcoming) up
   else: combine head with upcoming (if any)

Interface: $\hbar$ as transduction
Basic idea: Lexical items marked + must associate with X0 stem
If selector is marked +, move there.
In BU transduction: 'Look ahead' to selector.
Strategy of precise definition:
a. On path from root to leaf, pass "+" to children
b. On way up, at each node:
if "+": then delete head and pass it (and upcoming) up
else: combine head with upcoming (if any)

One simple rule suffices for all our examples

Interface: Linear ordering o

A linearized MG = (MG, o).

o_{svo}: first merge $<$; nonfirst $>$

o_{sov}: all $>$

Both: Very minor variations on ℓ.
Like first/nonfirst in SVO, easy to add category-sensitivity.

Interface: Linear ordering o
A linearized MG = (MG, o).
o_{svo}: first merge $<$; nonfirst $>$
o_{sov}: all $>$
Both: Very minor variations on ℓ.
Like first/nonfirst in SVO, easy to add category-sensitivity.

Chomsky (1995:340) "we take the LCA to be a principle of the phonological component"
Chomsky&al (2019:4) "a matter of externalization of internally generated expressions"

Still no consensus on constituent order:
cf. LCA vs. Abels&Neeleman 2012 and many others

Interface: o_{svo}

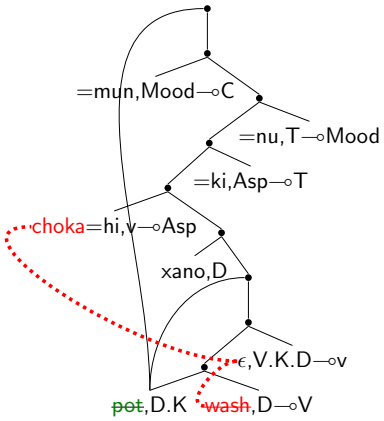
o_{svo} lex = { (lex,label) }

o_{svo} {SO_1,...,SO_i} =
  find unique negWS among ℓSO_1,...,ℓSO_i; #ineff!
  if negWS lexical,
  then: if >1 pos f:
        then head(negWS)
        else head(negWs):heads(posWSs)
  else: if >1 pos f:
        then head(negWS)
        else heads(posWSs) ++ [head(negWS)]

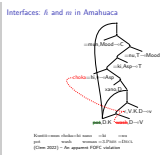
Inefficient: While ℓ goes to leaves and then applies d on the way up, this one goes to leaves, and then on its way up, repeatedly calls ℓ which goes to the leaves again to compute the label and workspace.

This inefficiency can be repaired when language model is reformulated as a single transduction – see last slide

Interfaces: $\hbar$ and m in Amahuaca



Kuntii=mun choka=hi xano =ki =nu
pot wash woman =3.Pres =Decl
(Clem 2022) – An apparent FOFC violation

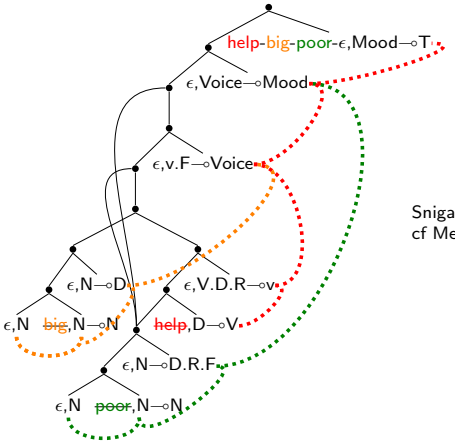


We distinguish head movement from 'm-merger' or 'amalgamation', which can form words based on adjacency/c-command – elements that need not be heads selected along a single extended projection (Harizanov&Gribanova 2019, Branigan 2023, inter alia)

Here a head movement is shown in red. After pot/kuntii, linearized as the left child of the root, it forms a phonological word with the clitic *mun*.

Currently thinking m could be formalized as (a component of) the map from linearized trees to prosodic structure (compare Stabler&Yu 2023)
** work in progress **

Interfaces: $\hbar$ and m in Aleut

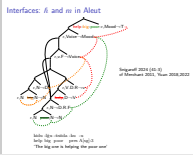


Snigaroff 2024 (41-3)
cf Merchant 2011, Yuan 2018,2022

kidu -lĝu -x̂siida -ku -u
help big poor pres A[sg]:3
'The big one is helping the poor one'

2024-09-02

└Interfaces: $\hbar$ and m in Aleut



More complicated cases like this one are appearing in the literature.

Cf. e.g. Branigan 2023, Oxford 2014 on
the Algonquian language Innu-aimûn, inter alia

The language model

Given any set (of sets...) of lexical items, we label and then slightly transform the structure as follows:

first ℓ:SO→WS

then to head of output WS, $\hbar$: SO→SO

then o: SO→(ordered) tree

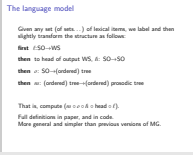
then m: (ordered) tree→(ordered) prosodic tree

That is, compute $(m \circ o \circ \hbar \circ \text{head} \circ \ell)$.

Full definitions in paper, and in code.
More general and simpler than previous versions of MG.

2024-09-02

└The language model



The full paper defines more modules –
for affix hopping, agreement, case, etc.

The language model as a single transduction

Better idea?: interleave steps and traverse SO once:

When ℓ applies d, apply interface functions immediately

Breaches modularity: interfaces become pre- and post-syntactic bc they affect SO, which d tests, breaking connectedness

Idea – apply not to results of d, but at t.
't' for 'transfer'
(still breaks connectedness: Collins&Stabler'16,§11)

work in progress . . .

The language model as a single transduction

Better idea?: interleave steps and traverse SO once:
When ℓ applies d, apply interface functions immediately
Breaches modularity: interfaces become pre- and post-syntactic bc they affect SO, which d tests, breaking connectedness
Idea – apply not to results of d, but at t.
't' for 'transfer'
(still breaks connectedness: Collins&Stabler'16,§11)
work in progress . . .

Given (3) on slide 11, the "Fundamental Principle of Grammar", it should be easy to fold all the interfaces together. . . (cf Kobele'11, Graf'11 on regularity of MG derivation languages)

So here we come close to Collins&Stabler'16. A number of Chomskian, minimalist proposals are trying to do closely related things informally.

Collins&Stabler §11 note problems that the breach causes for remnant movement, since that is a case where an element in the workspace can be complete, dropped from the workspace by t, even when it contains another element that is not inert.

But it is not obvious that the problem is restricted to that case. We need to watch for anything d tests that is possibly affected by interfaces, and for any information carried to the leaves top-down (as in the proposed treatment of head movement)

If we want incremental structure building, then we want to fold together all interface transductions, not just those that are regarded as part of 'transfer'

A Merge and label modularized: A complete Haskell implementation on 1 page

This implements our modular definitions of merge and labeling, the binary formulation. For all our definitions, many examples that use them, and a Python version, see https://github.com/epstabler/mgt.

```
module MgBin where
import Data.MultiSet (MultiSet, toList, fromList) -- Multiset needed. E.g., use: ghci -package multiset
import Data.List ( partition )
import Data.Bifunctor (Bifunctor, bimap, first, second)

type Label = ([String], [String])
type Lex = ([String], Label)
data SO = L Lex | S (MultiSet SO) | O PhTree deriving (Show, Eq, Ord)
type WS = ([SO],[Label])
data PhTree = Pl Lex | Ps [PhTree] | Pz deriving (Show, Eq, Ord)

-- merge a sequence of SOs into one SO, one multiset
mrg :: [SO] -> SO
mrg sos = S (fromList sos)

-- append two pairs of lists, coordinate-wise
(+++) :: Bifunctor bf => ([a1], [a2]) -> bf [a1] [a2] -> bf [a1] [a2]
(xs,ys) +++ pairOfLists = bimap (xs++) (ys++) pairOfLists

-- partition a pair of lists (xs,ys) according to whether each (x_i,y_i) has property p
ppartition :: (a1 -> a2 -> Bool) -> ([a1], [a2]) -> (([a1], [a2]), ([a1], [a2]))
ppartition _ ([],[]) = (([],[]),([],[]))
ppartition p (x:xs, y:ys) = let (ps,nonps) = ppartition p (xs,ys) in
  if p x y then (bimap (x:) (y:) ps, nonps) else (ps, bimap (x:) (y:) nonps)

-- partition sequence of WSs to separate WS elements that have a negative element
wssNeg :: [WS] -> ([WS],[WS])
wssNeg = partition ((/= []).fst.head.snd)

-- partition WS elements to separate WS elements with labels beginning with positive feature f
wsPosMatch :: String -> WS -> (WS,WS)
wsPosMatch f = ppartition (\_ y -> ((== f).head.snd) y)

-- check/delete already matched features of head and complement
ck :: [Label] -> [Label]
ck [h,c] = [ first tail h, second tail c ]

-- delete inert, trival elements of a workspace
t :: WS -> WS
t = snd. ppartition (\_ y -> y == ([],[]))

-- partition list of workspaces into WS with neg f and matching pos f's, and WS of non-matching elements
match:: [WS] -> (WS,WS)
match wss = let ([(so:sos, label : labels )], poswss) = wssNeg wss in
  let f = (head. fst) label in case (wsPosMatch f (sos, labels), poswss) of
      ((([so'],[label']), imOthers), [(so'':_,_)]) ->          -- IM
        if so'' == so' then ( ([so,so'],[label, label']), imOthers ) else error "merge-over-move"
      ((([],[]), imOthers), [ws]) -> case wsPosMatch f ws of  -- EM
        (([so'],[label']), emOthers) -> ( ([so,so'],[label, label']), imOthers +++ emOthers )

-- return workspace if it respects smc, else error
smc :: WS -> WS
smc (sos, labels) = if smc' [] labels then (sos, labels) else error "smc violation" where
    smc' _ [] = True
    smc' sofar ((([],f:_): labels) = (f `notElem` sofar) && smc' (f: sofar) labels
    smc' sofar (_: labels) = smc' sofar labels

-- derivational step, derives WS with a new merged SO and its new label
d :: [WS] -> WS
d wss = let ((sos, labels), others) = match wss in smc (t (mrg sos: tail sos, ck labels) +++ others)

-- extend d (partially) through the domain of merge
ell :: SO -> WS
ell (L lx) = ([L lx], [snd lx])
ell (S s) = d (map ell (toList s))
```

A.1.1. Haskell: Head movement

```
module MgBinH where
import Data.MultiSet (MultiSet, fromList, toList) -- Multiset needed. E.g. use: ghci -package multiset
import MgBin (SO(S,L), ell, wssNeg, wsPosMatch)

-- map strings [w,...] to number of head-incorporator +'s on w, else 0
inc :: [String] -> Int
inc s = case s of { (('+':s'):_) -> 1 + inc [s'] ; _ -> 0 }

-- where i = #heads needed by c-commanding selector, (h i so) = (heads, so')
h :: Int -> SO -> ([String], SO)
h 0 (L lex) = ([], L lex)
h 1 (L (w,fs)) = (w, L ([],fs))
h i (S s) = let ([nws],pws:pwss) = wssNeg (map ell (toList s)) in case (head.fst) nws of
      L (w,fs) -> let i' = inc w + max 0 (i-1) in
          let (hs,pso) = h i' ((head.fst) pws) in case i of
            0 -> ([], S (fromList (L (hs ++ w, fs) : pso : [])))
            1 -> (hs ++ w, S (fromList (L ([], fs) : pso : [])))
            _ -> (w ++ hs, S (fromList (L ([], fs) : pso : [])))
      nso -> let (hs,nso') = h i nso in
          let psos = map (head.fst) (pws:pwss) in
            (hs, S (fromList (nso' : psos)))
```

A.1.2. Haskell: Ordering

```
module MgBinO where
import Data.MultiSet (MultiSet, toList) -- Multiset needed. E.g. use: start ghci -package multiset
import MgBin (SO(S,L,O), PhTree(Pl,Ps), WS, ell, wssNeg, wsPosMatch)

o_svo :: SO -> SO
o_svo (O t) = O t
o_svo (L (w,l)) = O (Pl (w,l))
o_svo so = let (nso, pso, _, posfs) = o so in case (o_svo nso, o_svo pso, posfs) of
                (O (Pl i),O t,_:_:_) -> O (Ps [Pl i, silent t])  -- first merged, moving
                (O (Pl i),O t,_)    -> O (Ps [Pl i, t])          -- first merged
                (O t,O t',_:_:_)    -> O (Ps [silent t', t])     -- non-first merged, moving
                (O t,O t',_)        -> O (Ps [t', t])            -- non-first merged

o_sov :: SO -> SO
o_sov (O t) = O t
o_sov (L (w,l)) = O (Pl (w,l))
o_sov so = let (nso, pso, _, posfs) = o so in case (o_sov nso, o_sov pso, posfs) of
                (O t,O t',_:_:_) -> O (Ps [silent t', t])
                (O t,O t',_)     -> O (Ps [t', t])

-- reveal what order depends on: (head SO, comp SO, pos head features, pos comp features)
o :: SO -> (SO,SO,[String],[String])
o (S s) = let ([nws],[pws]) = wssNeg (map ell (toList s)) in
  let (so:sos,(f:_,_):labels) = nws in case wsPosMatch f (sos, labels) of
    (([so'],[label']), _) -> ((head.fst) nws, (head.fst) pws, (snd.head.snd) nws, snd label')
    _ ->                     ((head.fst) nws, (head.fst) pws, (snd.head.snd) nws, (snd.head.snd) pws)

-- tentatively, for readability: instead of deleting moved PhTree, parenthesize its yield
silent :: PhTree -> PhTree
silent (Pl (ws,label)) = Pl (map (\w -> if head w == '(' && last w == ')' then w else "(" ++ w ++ ")") ws, label)
silent (Ps phs) = Ps (map silent phs)
```

A.1.3. Haskell: M-merger

```
module MgBinM where
import  MgBin (SO(S,L,O), PhTree(Pl,Ps,Pz))

data  M = Mup | Mdn | Mz deriving  (Show, Eq,  Ord)

-- parse [w ,...]   to  Mup, Mdn or Mz, based on  features   marked on w,   if   any
mck ::  [ String ]  -> M
mck [] = Mz
mck (w:_) =  if   length   w > 0  then  case   last   w of  {  '^'  -> Mup ; '~'  -> Mdn ; _ -> Mz } else Mz

-- (m mc ph) = (head,  ph')  where mc = mck of  immediate  c-commander
m ::  M -> PhTree -> (PhTree, PhTree)
m Mz (Pl lex )  = (Pz,  (Pl  lex ))

{-
-- where i  = #heads needed by  c-commanding selector,  (h  i  so)  = (heads,  so' )
h  ::   Int   -> SO -> ([String], SO)
h  0  (L  lex )  = ([] , L lex )
h  1  (L  (w,fs ))  = (w,  L  ([] ,fs ))
h  i  (S  s)  =  let   ([nws],pws:pwss) = wssNeg (map ell   (toList   s)) in case (head.fst) nws of
        L  (w,fs )  -> let  i'  =  inc   w + max 0 (i-1) in
              let   (hs,pso)  = h  i'   ((head. fst )  pws)  in  case   i   of
                0  -> ([] , S (fromList   (L (hs ++ w, fs) : pso : [] )))
                1  -> (hs ++ w, S (fromList   (L ([] , fs) : pso : [] )))
                _  -> (w ++ hs, S (fromList   (L ([] , fs) : pso : [] )))
        nso  -> let  (hs,nso' )  = h  i  nso  in
              let   psos  = map (head. fst )  (pws:pwss)  in
                (hs,   S  (fromList   (nso' : psos)))

-}
```