Modular Minimalist Grammar

Edward Stabler UCLA

2024-05-25

- 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.

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

Modular Minimalist Grammar
Edward Stabler UCLA
2024-05-25
• 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.
Thanks to audiences at the MG+1, MG+2 meetings for helpful suggestions!

MG derivational steps are broken up then reassembled,

This 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! The scope 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:

$$([1,2],[a,b]) \text{ +++ } ([3],[c]) = ([1,2,3],[a,b,c])$$

(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])

<u>One</u> 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:
  SO = g | {SO}

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

Our first reformulation of MG will derive workspaces.

2024-05-25

└Setup

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:
SO = g | {SO}

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

Our first reformulation of MG will derive workspaces.

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

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 the same derived structures as in earlier MGs.)

5 linguistic functions

$$(\mathrm{match})\ \frac{\mathsf{WS}_1,\ldots,\mathsf{WS}_i \quad \text{initially, for } i\in\{1,2\}}{\text{matching WS, non-matching WS}}$$

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

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

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

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

2024-05-25

└5 linguistic functions

5 linguistic functions

(match) WS₁,...,WSᵢ initially, for i ∈ {1,2} / matching WS, non-matching WS
(mrg) SO₁,...,SOₙ / {SO₁,...,SOₙ}
(ck) f.α ⊸ β₁, f.β₂ / α ⊸ β₁, β₂
(t) WS / WS - (SOᵢ, labelᵢ) pairs where labelᵢ empty
(smc) WS / WS if no 2 positive labels have same 1st feature

Linearization is left to PF interface...
  (collapsing what is sometimes called ‘merge1’ and ‘merge2’)

match combines inputs then splits into matching and non-matching part
  returning matching WS with head first.
It also enforces move-over-merge: ∃!1 WS if ‘move’ is possible.

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

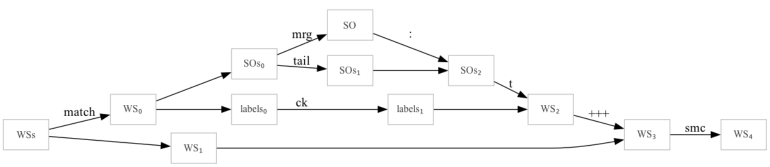
Function t removes the inert, ‘trivial’ SOs, those with no features...
  (collapsing ‘merge1/2’ and ‘merge3’, and ‘move1’ and ‘move 2’)
  An example will make this clear, below

MG composed: The derivational step

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

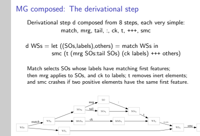
d WSs = let ((SOs,labels),others) = match WSs in
smc (t (mrg SOs:tail SOs) (ck labels) +++ others)

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.



2024-05-25

└MG composed: The derivational step



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

The python implementation on github 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) ) ).ppend(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. e.g. https://en.wikipedia.org/wiki/Let_expression and Davies & Pfenning (2001)

MG composed: The derivational step

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

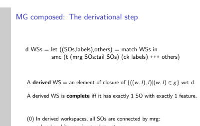
A **derived** WS = an element of closure of $\{((w, l), l) | (w, l) \in g\}$ wrt d.

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 its non-inert substructures.

2024-05-25

└MG composed: The derivational step

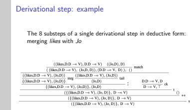


(0) means that we can view the workspace as a moving, labeled ‘window’ on the head SO, a kind of ‘locus of attention’

Derivational step: example

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

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



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:

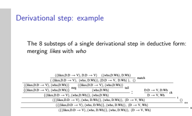
(praise, $D.D \multimap V$)	Given 2 DPs, a VP.
	Dot for conjunction
	Empty conjunction written $\top$, for top, true
	Antecedent features are 'negative'
(which, $N \multimap D.Wh$)	Given NP, a wh DP.
(student, $\top \multimap N$)	A noun(phrase).
	Usually written: (student, N)
$\top$	A label with no features.

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*

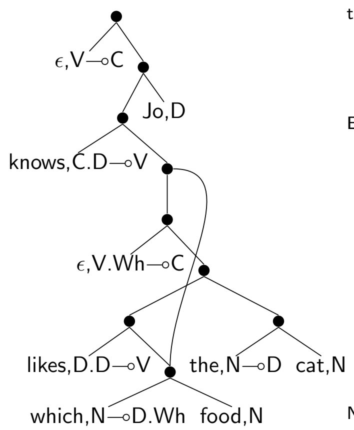
$$
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{((\text{likes},D.D \multimap V),\ D.D \multimap V) \quad ((\text{who},D.Wh),\ D.Wh)}{(\ (\text{likes},D.D \multimap V),\ (\text{who},\ D.Wh)),\ (D.D \multimap V,\ D.Wh)\ ),\ \ ()}\ \text{match}
}{
\dfrac{
\dfrac{((\text{likes},D.D \multimap V),\ (\text{who},D.Wh))}{\{(\text{likes},D.D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



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,

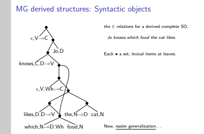
Jo knows which food the cat likes.

Each • a set; lexical items at leaves.

Now, easier generalization. . .

2024-05-25

└MG derived structures: Syntactic objects

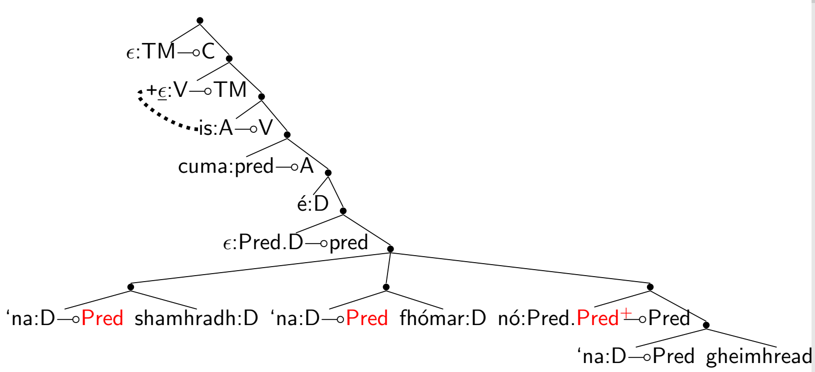


Since the feature checking done by d is so simple, after some practice, I find the SO itself to be the most readable notation for the derivation, though I can understand why linguists prefer highly redundant X-bar-like notations in the literature.

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

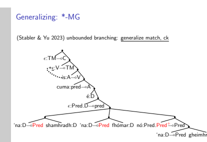
Generalizing: *-MG

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



2024-05-25

└Generalizing: *-MG



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*.

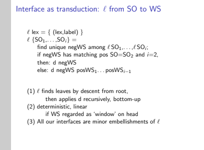
Using multisets is essentially equivalent to indexing, but local.

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

Interface as transduction: ℓ from SO to WS

ℓ lex = { (lex,label) }
ℓ {SO_1,...,SO_i} =
    find unique negWS among $\ell\,SO_1,\ldots,\ell\,SO_i$;
    if negWS has matching pos SO=SO_2 and i=2,
    then: d negWS
    else: d negWS $posWS_1 \ldots posWS_{i-1}$

(1) ℓ finds leaves by descent from root,
    then applies d recursively, bottom-up
(2) deterministic, linear
    if WS regarded as 'window' on head
(3) All our interfaces are minor embellishments of ℓ



Before defining a transduction in any new notation, it is good practice to define a very simple one, like the identity transduction. Here, instead, we define the mapping from SO to its workspace (if it has one). At each point, if IM, and the sister is in fact the element moving from inside the head: apply d to the negative argument. Else EM: apply d to all the arguments.

(3) is an important idea.

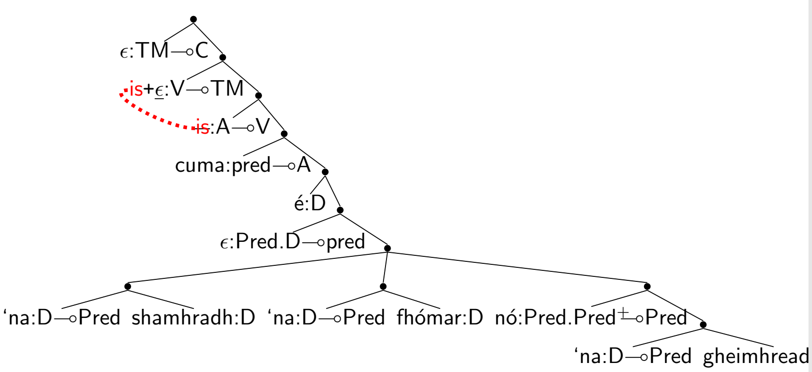
Cf. Hornstein's (2024, pp7-8) "Extended Merge hypothesis", the "Fundamental Principle of Grammar": "All grammatical relations are merge-mediated."

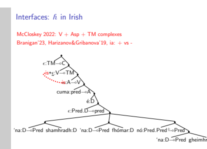
Hornstein attributes this idea to Epstein 1999, Collins 2007, but I think essentially this same idea is implicit in much of CCG, LFG, HPSG, etc too

Interfaces: $\hbar$ in Irish

McCloskey 2022: V + Asp + TM complexes

Branigan'23, Harizanov&Gribanova'19, ia: + vs -

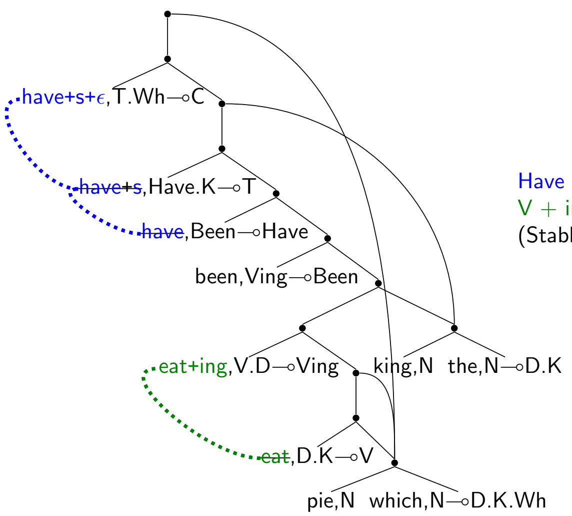




V raises to TM

McCloskey 2022 discusses some more complex examples,
left for future work

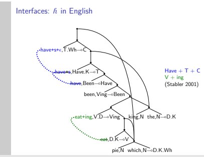
Interfaces: $\hbar$ in English



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

2024-05-25

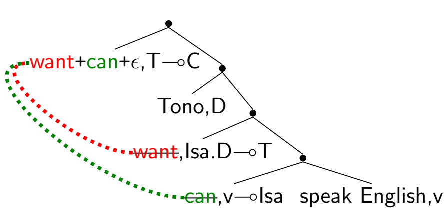
└Interfaces: $\hbar$ in English



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

which pie have $+$s $+\epsilon$ the king been eat $+$ing

Interfaces: $\hbar$ in Javanese



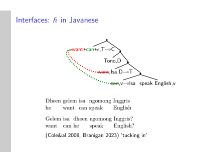
Dheen	gelem	isa	ngomong	Inggris
he	want	can	speak	English

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

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

2024-05-25

└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

Interface: h 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: h as transduction

Interface: h 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: 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

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

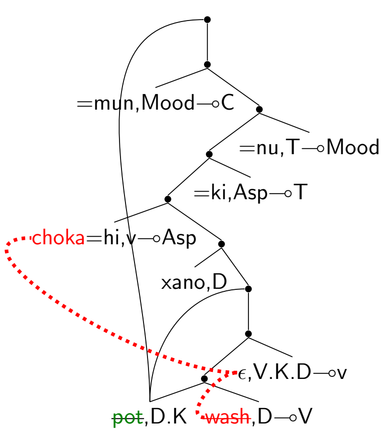
Interface: o_{svo}

o_{svo} lex = { (lex,label) }
o_{svo} $\{SO_1,\ldots,SO_i\}$ =
  find unique negWS among $\ell\,SO_1,\ldots,\ell\,SO_i$;   #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



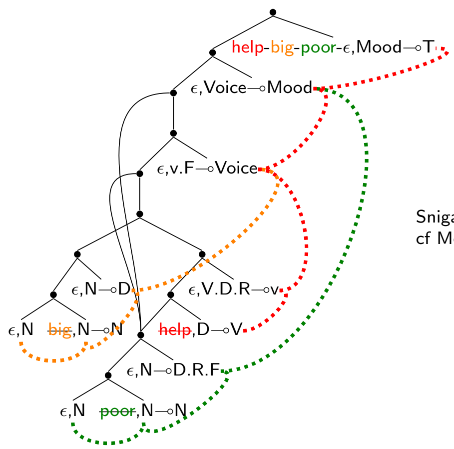
Kuntii=mun choka=hi xano =ki =nu
pot wash woman =3.PRES =DECL
(Clem 2022) – An apparent FOFC violation

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, but notice e.g. that pot/kuntii, linearized as the left child of the root, forms a word with =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

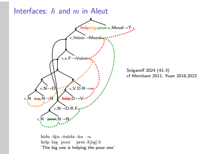


Snigaroff 2024 (41-3)
cf Merchant 2011, Yuan 2018,2022

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

2024-05-25

└Interfaces: $\hbar$ and m in Aleut



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

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

The language model

Given any set (of sets. . .) of lexical items:

first ℓ:SO→WS

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

then o: SO→(ordered) tree

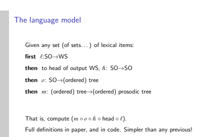
then m: (ordered) tree→(ordered) prosodic tree

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

Full definitions in paper, and in code. Simpler than any previous!

2024-05-25

└The language model



The language model as a single transduction

Better idea?: interleave steps and traverse SO <u>once</u>:

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. (But I like to have crisp, formal definitions <u>and</u> a running implementation to avoid having to do so many tedious checks by hand!)

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

A Implementations: Haskell

A.1 Haskell: Derivations with unbounded merge

```haskell
-- https://github.com/epstabler/mgt/tree/main/haskell/Mg/Mg.hs
module Mg where     -- Multiset needed. E.g., start ghci with: stack ghci --package multiset
import Data.MultiSet (MultiSet)
import qualified Data.MultiSet as MultiSet
import Data.List ( partition )

data F = C | D | N | V | A | P | Wh | Pred | Predx | T | K | Vx | Scr | Modal | Have | Be | Been |
         Ving | Ven | Lf | Rt | B | Visa | Vgelem deriving (Show, Eq, Ord)
data Ft = One F | Plus F deriving (Show, Eq, Ord)
type Label = ([Ft], [Ft])
type Lex = ([String], Label)
data PhTree = Pl Lex | Ps [PhTree] deriving (Show, Eq, Ord)
data SO = L Lex | S (MultiSet SO) | O PhTree deriving (Show, Eq, Ord)
type WS = ([SO],[Label])

-- basics: pair cons, pair concatenation, pair partition, feature parser
(a,b) @@ (x,y) = (a:x, b:y)
(x,y) +++ (z,w) = (x ++ z, y ++ w)
ppartition _ ([],[]) = (([],[]),([],[]))
ppartition p (f:fs, s:ss) = let (ps,nonps) = ppartition p (fs,ss) in
  if p f s then ((f,s) @@ ps, nonps) else (ps, (f,s) @@ nonps)
fplus ft = case ft of (One f) -> (f, False); (Plus f) -> (f, True)

-- merge
mrg :: [SO] -> SO
mrg sos = S (MultiSet.fromList sos)

-- already matched features can be 'forgotten'
ck :: [Label] -> [Label]
ck ((_:nns,nps):([],_:pps):more) = [(nns,nps), ([],pps) ] ++ map (const ([],[])) more

-- constituents 'forgotten' from workspace when 'inert', i.e. all features 'forgotten'
t :: [SO] -> [Label] -> WS
t (_:sos) (([],[]):labels) = t sos labels
t (so:sos) (label:labels) = (so, label) @@ t sos labels
t [] [] = ([], [])

match:: [WS] -> (WS,WS)
match wss =
  let ([(so:sos, label:labels)], poswss) = partition ((/= []).fst.head.snd) wss in  -- partition neg WSs
  let (f,plus) = (fplus.head.fst) label in
    case (ppartition (\x y -> ((== One f).head.snd) y) (sos, labels), poswss) of  -- partition matches
      ((([so'],[label']), imOthers), []) -> (([so,so'],[label,label']), imOthers)           -- IM
      ((([],[]), imOthers), ws:wss') -> case ppartition (\x y -> ((== One f).head.snd) y) ws of -- EM
        (([so'],[label']), emOthers) ->
          if plus && emOthers == imOthers
          then ( ([so,so'],[label,label']) +++ atb label' emOthers wss', imOthers)
          else if null wss' then ( ([so,so'],[label,label']), imOthers +++ emOthers) else error "match"
  where
    atb :: Label -> WS -> [WS] -> WS -- collect additional comps with same label and movers
    atb _ _ [] = ([],[])
    atb l movers (ws:wss) =
      let (([so'],[label']), others) = ppartition (\x y -> y == l) ws in -- partition matches
        if others == movers then ((so', label') @@ (atb l movers wss)) else error "match: ATB error"

-- if labels of WS satisfy shortest move constraint, return WS; else crash --'too much to remember!'
smc :: WS -> WS
smc (sos, labels) = if smc' [] labels then (sos, labels) else error "smc violation" where
  smc' _ [] = True
  smc' sofar (([],p:ps): labels) = let (f,_) = fplus p in if f `elem` sofar then False else smc' (f:sofar) labels
  smc' sofar (_: labels) = smc' sofar labels

-- the derivational step: binary merge and label
d :: [WS] -> WS
d wss = let ((sos, labels), others) = match wss in
  smc (t (mrg sos: tail sos) (ck labels) +++ others)
```

A.2. Haskell: Derivation as transduction

```
-- https://github.com/epstabler/mgt/tree/main/haskell/Mg/MgL.hs
module MgL where -- Multiset needed. E.g., start ghci with: stack ghci --package multiset
import Data.MultiSet (MultiSet)
import qualified Data.MultiSet as MultiSet
import Data.List ( partition )
import Mg

ell :: SO -> WS
ell (L lex) = ([L lex], [snd lex])
ell (S s) = let ([nws],pws:pwss) = partition ((/= []).fst.head.snd) (map ell (MultiSet.toList s)) in -- partition neg WS
  let (so:sos,(f:_,_):labels) = nws in case ppartition (\x y -> ((==f).head.snd) y) (sos, labels) of -- partition matches
    (([so'],_), _) -> if ( (head.fst) pws /= so' || not (null pwss) ) -- IM
                      then error "ell: merge-over-move violation"
                      else d [nws]
   _ -> d (nws:pws:pwss)                                         -- EM

-- Note how the recursive case of ell calls itself immediately, going right down to the leaves -- the base case.
-- Then, d is applied to build the workspaces bottom-up.
-- So this is a multi bottom-up transduction on unordered trees (i.e. on multisets).
```

A.3. Haskell: Head movement

```
-- https://github.com/epstabler/mgt/tree/main/haskell/Mg/MgH.hs
module MgH where -- Multiset needed. E.g., start ghci with: stack ghci --package multiset
import Data.MultiSet (MultiSet)
import qualified Data.MultiSet as MultiSet
import Data.List ( partition )
import Mg
import MgL

-- 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) = partition ((/= []).fst.head.snd) (map ell (MultiSet.toList s)) in -- partition neg WS
    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
        let psos = atbh i' hs pwss in
          if i == 0
          then ([], S (MultiSet.fromList (L (hs ++ w, fs) : pso : psos)))
          else
            if i == 1
            then (hs ++ w, S (MultiSet.fromList (L ([], fs) : pso : psos)))
            else (w ++ hs, S (MultiSet.fromList (L ([], fs) : pso : psos)))
      nso ->
        let (hs,nso') = h i nso in
        let psos = map (head.fst) (pws:pwss) in
          (hs, S (MultiSet.fromList (nso' : psos)))
  where
    atbh :: Int -> [String] -> [WS] -> [SO] -- collect additional comps with hs extracted
    atbh _ _ [] = []
    atbh i hs (pws:pwss) = let (hs', pso) = h i ((head.fst) pws) in
        if hs' == hs then pso:atbh i hs pwss else error "atbh error"
```

A.4. Haskell: Ordering

```
-- https://github.com/epstabler/mgt/tree/main/haskell/Mg/MgO.hs
module MgO where -- Multiset needed. E.g., start ghci with: stack ghci --package multiset
import Data.MultiSet (MultiSet)
import qualified Data.MultiSet as MultiSet
import Data.List ( partition )
import Mg
import MgL
import MgH

o_svo :: SO -> SO
o_svo (O t) = O t -- NB! recurse only as deeply as necessary
o_svo (L (w,l)) = O (Pl (w,l))
o_svo so = let (nso, pso, _, posfs, pwss) = o so in
           let psos = map (head.fst) pwss in
           let ts = map (\x -> case o_svo x of (O t) -> t) psos in
             case (o_svo nso, o_svo pso, posfs) of
               (O (Pl i),O t,_:_:_) -> O (Ps [Pl i, silent t])
               (O (Pl i),O t,_) -> O (Ps (Pl i:t:ts))
               (O t,O t',_:_:_) -> O (Ps [silent t', t])
               (O t,O t',_) -> O (Ps (t':(ts ++ [t])))

o_sov :: SO -> SO
o_sov (O t) = O t -- NB! recurse only as deeply as necessary
o_sov (L (w,l)) = O (Pl (w,l))
o_sov so = let (nso, pso, _, posfs, pwss) = o so in
           let psos = map (head.fst) pwss in
           let ts = map (\x -> case o_svo x of (O t) -> t) psos 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':ts)++[t]))

-- map SO to what ordering usually depends on: (head SO, comp SO, pos head features, pos comp features, addtl pos WSs)
o :: SO -> (SO,SO,[Ft],[Ft],[WS])
o (S s) = let ([nws],pws:pwss) = partition ((/= []).fst.head.snd) (map ell (MultiSet.toList s)) in -- partition neg WS
  -- NB: to get pos features of IM complement, we need to find them in (ell nws); otherwise, they're in pws
  let (so:sos,(f:_,_): labels) = nws in case ppartition (\x y -> ((==f).head.snd) y) (sos, labels) of -- partition matches
    (([so'],[label']), _) -> ((head.fst) nws, (head.fst) pws, (snd.head.snd) nws, snd label', pwss)
    _ ->                     ((head.fst) nws, (head.fst) pws, (snd.head.snd) nws, (snd.head.snd) pws, pwss)

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)
```

B Implementations: Python

Type hints are added to allow each python file to be checked with mypy. See https://github.com/epstabler/mgt for this code and examples that use these functions, with command-line and nltk-based graphical display.

B.1. Python setup

Unlike Haskell, Python cannot have lexical items in a set if those items are built with lists of features. (Python lists are not hashable.) So lexical items are (SO, label) pairs, where label is also a pair (posFeatures, negFeatures), where both positive and negative features are given in tuples.

Python tuples are written with parentheses. Since parentheses are also used for grouping, an extra comma must be added to length 1 tuples for disambiguation. The empty sequence is ().

The file mgTests.py has functions to print our data structures in more readable form. So, for example, here is the tuple-based representation of the grammar from §1.1.2 of the paper, and the 'pretty-printed' version of that grammar:

```
> python
>>> from mgTests import *

>>> for i in g112: print(i)

 ((), (('V',), ('C',)))
 ((), (('V', 'Wh'), ('C',)))
(('Jo',), ((), ('D',)))
(('the',), (('N',), ('D',)))
(('which',), (('N',), ('D', 'Wh')))
(('who',), ((), ('D', 'Wh')))
(('cat',), ((), ('N',)))
(('dog',), ((), ('N',)))
(('food',), ((), ('N',)))
(('likes',), (('D', 'D'), ('V',)))
(('knows',), (('C', 'D'), ('V',)))

>>> ppMg(g112)

 (, V -o C)
 (, V.Wh -o C)
 (Jo, D)
 (the, N -o D)
 (which, N -o D.Wh)
 (who, D.Wh)
 (cat, N)
 (dog, N)
 (food, N)
 (likes, D.D -o V)
 (knows, C.D -o V)

>>>
```

Python also does not allow multisets of multisets, so an alternative implementation of multisets is provided here, based on frozendicts. The frozendict module can be installed with 'pip install frozendict'. That module is used in the definition of the class of SO objects.

The basic types of objects (LI, SO, Label, WS) are all defined in `mgTypes.py`. That file is not listed here, since it is more complex than the corresponding type definitions and functions in Haskell, which were listed in full in Appendix A. But once appropriate Python object classes are defined, the python definitions of MG functions are similar to the Haskell...

B.2. Python: Derivations with unbounded merge

```python
""" https :// github .com/ epstabler /mgt/ tree /main/python/mg.py """
from mgTypes import *    # defines classes of objects : LI, SO, Label, WS
from typing import Tuple # for type-checking with mypy

def mrg(seq: list) -> SO:
  """ merge: form multiset from sequence of SOs """
  return SO(seq)

def ck(labels: list) -> list:
  """ remove first features from labels """
  return [f.ck() for f in labels [0:2]] + [Label ((),()) for f in labels [2:]]

def t(ws: WS) -> WS:
  """ remove SOs with no features in their label """
  return ws.pfilter(lambda x: not(x[1].is_empty()))

def fplus( feature ) -> Tuple[str,bool]:
  """ parse feature """
  if feature [-1] == '+': return (feature[:-1], True)
  else: return (feature, False)

def match(wss:list) -> Tuple[WS,WS]:
  """ partition elements of elements of WSs into (matchingWS, non-matchingWS) """
  (negwss, poswss) = partition (lambda x: x.is_neg (), wss) ## partition neg WSs (def in mgTypes.py)
  if len(negwss) != 1: raise RuntimeError("match: too many neg workspaces")
  so0, sos0, label0, labels0 = negwss[0]._sos [0], negwss[0]._sos [1:], negwss[0]. _labels [0], negwss[0]. _labels [1:]
  (f, plus) = fplus ( label0 ._neg[0])
  (IMmatches, IMothers) = WS(sos0,labels0 ). ppartition (lambda x: x[1]._pos[0] == f) # partition matches
  if IMmatches._sos:
    if poswss != []: raise RuntimeError("match: too many im pos workspaces")
    so1, label1 = IMmatches._sos[0], IMmatches._labels[0]
    return ( WS([so0,so1], [label0 , label1 ]), IMothers )
  else:
    pws, pwss = poswss[0], poswss[1:]
    (EMmatches, EMothers) = pws. ppartition (lambda x: x[1]._pos[0] == f) # partition matches
    if EMmatches._sos:
      so1, label1 = EMmatches._sos[0], EMmatches._labels[0]
      if plus and str(IMothers) == str(EMothers): # str to avoid comparison issues
        moreComps = atb(label1, EMothers, pwss)
        return ( WS([so0,so1], [label0 , label1 ]). pappend(moreComps), IMothers )
      else:
        if pwss != []: raise RuntimeError("match: too many em pos workspaces")
        return ( WS([so0,so1], [label0 , label1 ]), IMothers.pappend(EMothers) )
    else: raise RuntimeError("match: no matching pos workspaces")

def atb( label , movers, wss) -> WS:
  additionalComplementWS = WS([],[])
  for ws in wss:
    (matches, others ) = ws. ppartition (lambda x: str(x[1]) == str(label)) # str to avoid comparison issues
    if len(matches._sos) == 1: # and others == movers:
      additionalComplementWS = additionalComplementWS.pappend(matches)
    else:
      raise RuntimeError("atb: non-matching pos workspaces")
  return additionalComplementWS

def smc(ws: WS) -> WS:
  """ if WS has no 2 pos labels with same 1st feature , return WS """
  sofar = []
  for label in ws._labels :
    if label . is_pos ():
      if label ._pos[0] in sofar : raise RuntimeError('smc violation blocked' )
      else: sofar . append( label ._pos [0])
  return ws

def d(wss:list) -> WS:
  """ the derivational step : given list of workspaces, return derived workspace """
  (matches, others ) = match(wss)
  return smc( t( WS( [mrg(matches._sos)] + matches._sos [1:], ck(matches. _labels ) ) ). pappend(others) )
```

B.3. Python: Derivation as transduction

```
""" https :// github.com/epstabler/mgt/tree/main/python/mgL.py """
from mg import * # this imports frozendict , mgTypes, mg

def ell(so):
  """ Map so to its derived workspace, if any.
   NB: For derived SOs, ell is recursively mapped to children , going right down to the leaves .
   Then, from the leaves , d is applied to build workspaces bottom-up.
   So this is a multi bottom-up transduction on unordered trees (i.e. on multisets ).
  """
  if isinstance(so,LI) or isinstance(so,O): return so.to_ws()
  else:
    (negwss, poswss) = partition (lambda x: x.is_neg (), map(ell,so. to_tuple ())) ## partition neg WSs (def in mgTypes.py)
    sos0, label0 , labels0 = negwss[0]._sos [1:], negwss[0]. _labels [0], negwss[0]. _labels [1:]
    (f, plus) = fplus ( label0 ._neg[0])
    (IMmatches, IMothers) = WS(sos0,labels0 ). ppartition (lambda x: x[1]._pos[0] == f) # partition matches
    if IMmatches._sos:
      if len(poswss) != 1 or \
         str(IMmatches._sos[0]) != str(poswss[0]._sos [0]): # str to avoid comparison issues
        raise RuntimeError("ell: move-over-merge error")
      return d( negwss )
    else:
      return d( negwss + poswss )
```

B.4. Interfaces in Python: Head movement

```
""" https :// github.com/epstabler/mgt/tree/main/python/mgH.py """
from mgL import *         # this imports frozendict , mgTypes, mg, mgL

def inc(h) -> int:
  """ return the number of head-incorporator '+' at the beginning of h """
  if not(h) or not(isinstance(h[0],str)):
    raise RuntimeError("inc: expected tuple of strings ")
  count = 0
  for i in range(len(h[0])):
    if h[0][i] == '+': count += 1
    else: break
  return count

def h(i, so):
  """ head movement, where i is the number of head-incorporators on governing head """
  if isinstance(so,LI):
    if i == 0: return ((), so)
    elif i == 1: return (so._ph, LI ((), so. _label . pair ()))
    else: RuntimeError("h: incorporator requirements not met")
  else:
    (negwss, poswss) = partition (lambda x: x.is_neg (), map(ell,so. to_tuple ())) ## partition neg WSs (def in mgTypes.py)
    so0 = negwss[0]._sos[0]
    pso0 = poswss[0]._sos[0]
    if isinstance(so0,LI):
      i0 = inc(so0._ph) + max([0,i-1])
      (hs,pso) = h(i0, pso0)
      psos = atbh(i0, hs, poswss[1:])
      if i == 0: return ((), SO([LI(hs + so0._ph, so0. _label . pair ()), pso] + psos))
      elif i == 1: return (hs + so0._ph, SO([LI ((), so0. _label . pair ()), pso] + psos))
      else: return (so0._ph + hs, SO([LI((), so0. _label . pair ()), pso] + psos))
    else:
      (hs,so1) = h(i, so0)
      psos = [ws._sos[0] for ws in poswss]
      return (hs, SO([so1] + psos))

def atbh(i, hs, wss) -> list:
  """ collect additional comps with hs extracted , across-the-board """
  if wss == []: return []
  else:
    (hs0, pso) = h(i, wss[0]._sos[0])
    if hs0 != hs: raise RuntimeError("atbh: non-identical head")
    else: return [pso] + atbh(i, hs, wss[1:])
```

B.5. Interfaces in Python: Linearization

```
""" https :// github .com/ epstabler / mgt/ tree / main/python/mgO.py """
from mgH import * # this imports frozendict , mgTypes, mg, mgL, mgH

def o_svo(so) -> O:
  """ map so to ordered svo tuple """
  if isinstance(so,LI):
    return O((so,))
  else:
    (nso, pso, _, posfs , pwss) = o(so)
    nt , pt = o_svo(nso ), o_svo(pso)
    psos = [ws._sos[0] for ws in pwss]
    pts = tuple(map(o_svo, psos))
    if len(posfs) > 1:
      comps = silent (( pt ,) + pts )
    else:
      comps = (pt ,) + pts
    if isinstance(nso,LI):
      return O( (nt,) + comps )
    else:
      return O( comps + (nt,) )

def o_sov(so) -> O:
  """ map so to ordered sov tuple """
  if isinstance(so,LI):
    return O((so,))
  else:
    (nso, pso, _, posfs , pwss) = o(so)
    nt , pt = o_sov(nso ), o_sov(pso)
    psos = [ws._sos[0] for ws in pwss]
    pts = tuple(map(o_sov, psos))
    if len(posfs) > 1:
      comps = silent (( pt ,) + pts )
    else:
      comps = (pt ,) + pts
    return O( comps + (nt,) )

def o(so) -> tuple:
  """ maps so to (head, comp, head_pos_features , comp_pos_features, otherPosWSs) """
  # NB: to get pos features of IM complement, we need to find them in ell (nws); otherwise , they're in pws
  if not(isinstance(so._so,frozendict.frozendict)): raise TypeError("o: type error ")
  (negwss, poswss) = partition (lambda x: x.is_neg (), map(ell,so. to_tuple ())) ## partition neg WSs (def in mgTypes.py)
  so0, sos0, label0 , labels0 = negwss[0]._sos [0], negwss[0]._sos [1:], negwss[0]. _labels [0], negwss[0]. _labels [1:]
  (f, plus ) = fplus ( label0 ._neg[0])
  (IMmatches, IMothers) = WS(sos0,labels0 ). ppartition (lambda x: x[1]._pos[0] == f) # partition matches
  if IMmatches._sos:
    so1, label1 = IMmatches._sos[0], IMmatches._labels[0]
    return (so0, so1, label0 ._pos, label1 ._pos, poswss[1:])
  else:
    so1, label1 = poswss[0]._sos [0], poswss[0]. _labels [0]
    return (so0, so1, label0 ._pos, label1 ._pos, poswss[1:])

def silent (x ):
  if isinstance(x,tuple):
    return tuple([silent(o) for o in x])
  elif isinstance(x,O):
    if len(x._tuple) == 1: # lexical item
      ph, fs = x._tuple [0]._ph, x._tuple [0]. _label . pair ()
      newph = tuple(['('+w+')' for w in ph])
      return O((LI(newph, fs),))
    else:
      return O( tuple([silent(o) for o in x._tuple]) )
```