Modular Minimalist Grammar

Edward Stabler UCLA

MG+2 2024

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

These slides, code (Haskell, Python), and (soon) draft paper:

https://github.com/epstabler/mgt

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.

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, ck, t, smc, match
+ 5 bureaucratic functions:

1 : [2,3] = [1,2,3]   aka ‘cons’

tail [1,2,3] = [2,3]

zip [(1,2), (3,4), (5,6)] = ([1,3,5], [2,4,6])

unzip ([1,3,5], [2,4,6]) = [(1,2), (3,4), (5,6)]

[1,2] ++ [3,4] = [1,2,3,4]   aka ‘concatenate’

One MG derivational step will be composed from these 10 simple functions. We cover all the subcases of earlier MG, but without breaking the rule up that way.

The 5 bureaucratic functions are familiar to programmers.

Setup

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

Syntactic object: a lexical item or set of syntactic objects:
  SO = g | {SO} | [SO]

Labeled syntactic object: a pair
  LSO = (SO,label)

Workspace: a set of LSOs:
  WS = {LSO}

Our first reformulation of MG will derive workspaces
– basically the same derived structures as in earlier MG

└Setup

Setup

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

Syntactic object: a lexical item or set of syntactic objects:
SO = g | {SO} | [SO]

Labeled syntactic object: a pair
LSO = (SO,label)

Workspace: a set of LSOs:
WS = {LSO}

Our first reformulation of MG will derive workspaces
– basically the same derived structures as in earlier MG

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

5 linguistic functions

$$\text{(match)}\ \frac{\mathsf{WS}_1,\ldots,\mathsf{WS}_i \quad \text{for } i \in \{1,2\}}{\text{head LSO:complement LSOs, movers}}$$

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

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

$$\text{(t)}\ \frac{\text{LSOs}}{\text{LSOs - `inert' LSOs, those with no features left}}$$

$$\text{(smc)}\ \frac{\mathsf{LSO}_0 \quad \ldots \quad \mathsf{LSO}_i}{\{\mathsf{LSO}_0,\ldots,\mathsf{LSO}_i\} \quad \text{if no 2 have same 1st feature}}$$

└5 linguistic functions

5 linguistic functions

(match) WS1,...,WSi for i ∈ {1,2} / head LSO:complement LSOs, movers
(mrg) SO1,...,SOn / {SO1,...,SOn}
(ck) f.α ⊸ β1, f.β2 / α ⊸ β1, β2
(t) LSOs / LSOs - 'inert' LSOs, those with no features left
(smc) LSO0 ... LSOi / {LSO0,...,LSOi} if no 2 have same 1st feature

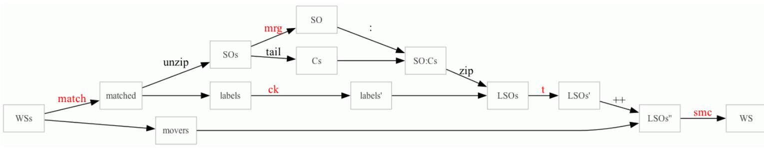
Linearization is left to PF interface. . .
  collapsing 'merge1' and 'merge2'

match pulls out LSOs with matching features, returning 2 sequences:
  matching elements with head first; remainder, aka 'movers'.
It also enforces move-over-merge: $\exists!1$ WS if 'move' is possible.

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

Function t removes 'trivial' LSOs, those with no features. . .
  This collapses 'merge1/2' and 'merge3', 'move1' and 'move 2'.
  An example will make this clear, below

MG composed: The derivational step



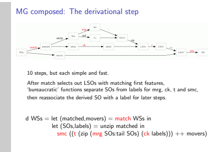
10 steps, but each simple and fast.

After match selects out LSOs with matching first features,
'bureaucratic' functions separate SOs from labels for mrg, ck, t and smc,
then reassociate the derived SO with a label for later steps.

d WSs = let (matched,movers) = match WSs in
let (SOs,labels) = unzip matched in
smc ((t (zip (mrg SOs:tail SOs) (ck labels))) ++ movers)

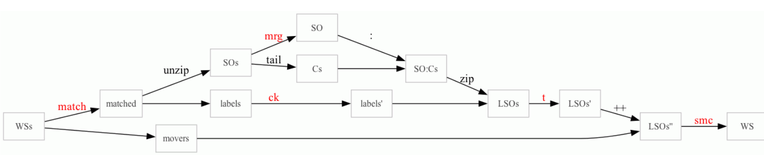
2024-03-31

└MG composed: The derivational step



This is the actual Haskell definition in my github implementation.

MG composed: The derivational step



d WSs = let (matched,movers) = match WSs in
let (SOs,labels) = unzip matched in
smc ((t (zip (mrg SOs:tail SOs) (ck labels))) ++ movers)

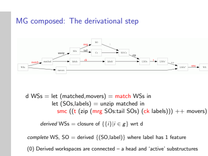
derived WSs = closure of $\{\{i\}|i \in g\}$ wrt d

complete WS, SO = derived {(SO,label)} where label has 1 feature

(0) Derived workspaces are connected – a head and 'active' substructures

2024-03-31

└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

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

└─Derivational step: example

Instead of merging a verb with a direct object in 1 specialized step, we do it in 10 simpler steps that cover all the cases.

Derivational step: example

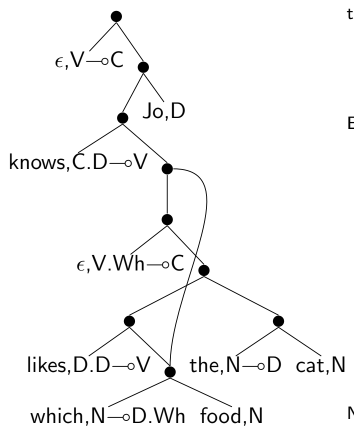
$$
\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\{((\mathrm{likes},D.D \multimap V), D.D \multimap V)\})\ \{((\mathrm{who},D.Wh), D.Wh)\}}{[\ ((\mathrm{likes},D.D \multimap V), D.D \multimap V), ((\mathrm{who},D.Wh), D.Wh)\ ],\ []}\ \mathrm{match}}{\dfrac{((\mathrm{likes},D.D \multimap V), (\mathrm{who},D.Wh))}{\{(\mathrm{likes},D.D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

└─Derivational step: example

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 10 steps

Since these 10 steps are exactly d, and d is the only rule, we can just show the 10 steps as one, and we do not need to label it!

MG derived structures: Syntactic objects



the $\in$ relations for a derived complete SO,

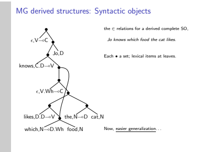
Jo knows which food the cat likes.

Each $\bullet$ a set; lexical items at leaves.

Now, easier generalization...

2024-03-31

└MG derived structures: Syntactic objects

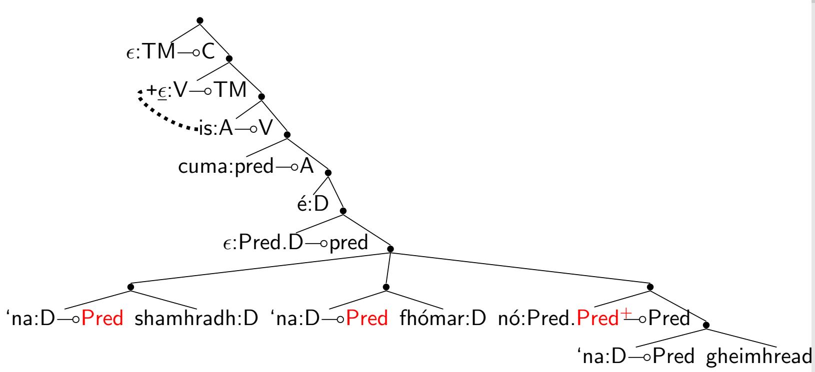


Since the feature checking done by d is so simple, after some practice, I find the SO itself to be the most readable notation for the derivation, though I can understand why linguists prefer highly redundant X-bar-like notations in the literature.

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

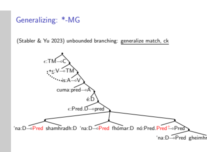
Generalizing: *-MG

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



2024-03-31

└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 {$\mathrm{SO}_1,\ldots,\mathrm{SO}_i$} =
  find unique negWS among $\ell\,\mathrm{SO}_1,\ldots,\ell\,\mathrm{SO}_i$;
  if negWS has matching pos SO=SO_2 and i=2,
  then: d negWS
  else: d negWS $\mathrm{posWS}_1\ldots\mathrm{posWS}_{i-1}$

(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 on syntactic objects. Here, instead of id, the mapping from SO to its workspace (if it has one). At each point, if IM, we check that the attached sister is in fact the element moving from inside the head, and apply d to the negative argument. Else EM: apply d to all the arguments.

(3) is an important idea.

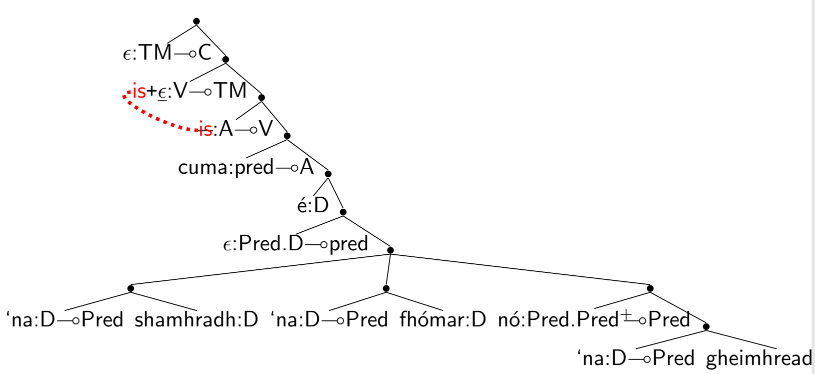
Cf. Hornstein’s (2024, pp7-8) “Extended Merge hypothesis”, the “Fundamental Principle of Grammar”: “All grammatical relations are merge-mediated.”

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

Cf. Milewski (2019,§§4.1,20.1) on ‘embellishment’ and Kleisli categories

Interfaces: $\hbar$ in Irish

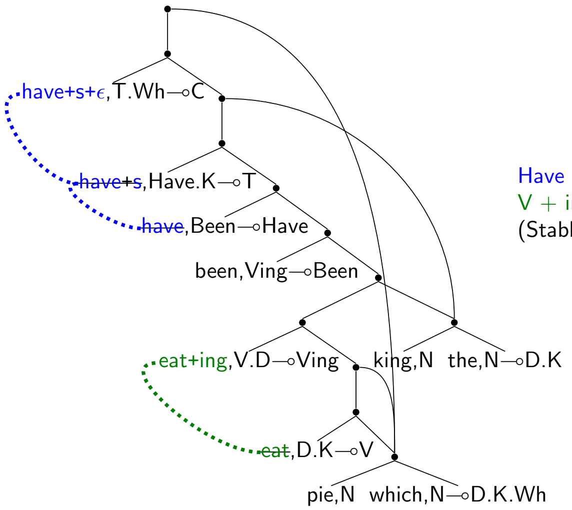
McCloskey 2022: V + Asp + TM complexes
Branigan’23, Harizanov&Gribanova’19, ia: + vs -



V raises to TM

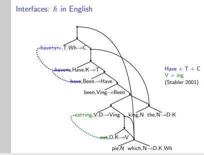
McCloskey 2022 discusses some more complex examples,
left for future work

Interfaces: $\hbar$ in English



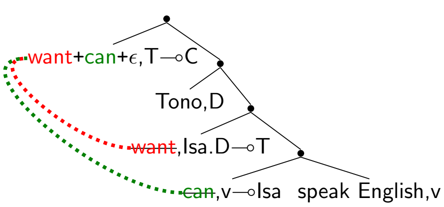
Have + T + C
V + ing
(Stabler 2001)

└Interfaces: $\hbar$ in English



The simplified treatment of Eng aux from Stabler 2001.
which pie have $+$s $+\epsilon$ the king been eat $+$ing

Interfaces: $\hbar$ in Javanese

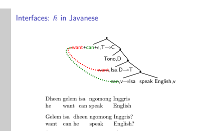


Dheen	gelem	isa	ngomong	Inggris
he	want	can	speak	English

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

(Cole&al 2008, Branigan 2023) ‘tucking in’

└Interfaces: $\hbar$ in Javanese



More than 1 verb can move up to C, ‘tucking in’ so that linear order corresponds to linear order of the linearized sources

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)

2024-03-31

└Interface: $\hbar$ as transduction

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: 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.

2024-03-31

└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"

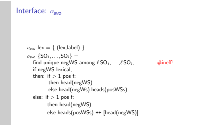
Still no consensus on constituent order:
cf. LCA vs. Abels&Neeleman 2012 and many others

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)]

2024-03-31

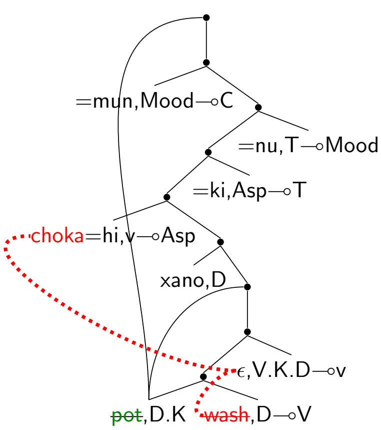
└Interface: o_{svo}



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

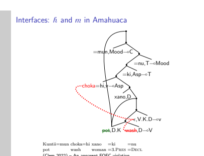


Kuntii=mun	choka=hi	xano	=ki	=nu
pot	wash	woman	=3.PRES	=DECL

(Clem 2022) – An apparent FOFC violation

2024-03-31

└Interfaces: $\hbar$ and m in Amahuaca

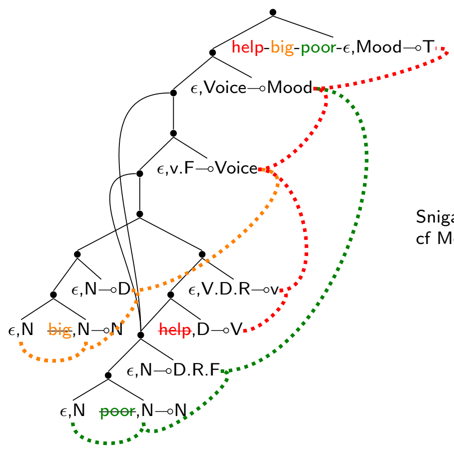


We distinguish head movement from ‘m-merger’ or ‘amalgamation’, which can form words based on adjacency/c-command – elements that need not be heads selected along a single extended projection (Harizanov&Gribanova 2019, Branigan 2023, inter alia)

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

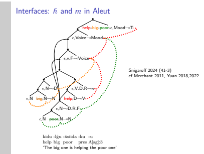


Snigaroff 2024 (41-3)
cf Merchant 2011, Yuan 2018,2022

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

2024-03-31

└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

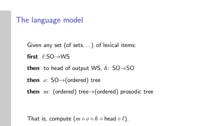
then m: (ordered) tree→(ordered) prosodic tree

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

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

2024-03-31

└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'15,§11)

work in progress! . . .

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'15,§11)
work in progress! . . .

Given (1), the "Fundamental Principle of Grammar", 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'15. 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. E.g. if heads are found by size, then deletion of structure by linearization can be problematic, etc, etc

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

One goal of the present project is to simplify the definition of minimalist grammars, and that goal should really be assessed by looking at a complete, formal statement. Haskell stays close to higher order logic and provides static typing which helps to keep things simple. Each function definition is preceded by a declaration of its type.

See https://github.com/epstabler/mgt for this code and examples that use these functions.

A.1. Haskell: Derivations with binary merge

```
module MgBin where
import Data.Set (Set)
import qualified Data.Set as Set
import Data.List
import qualified Data.List as List

data Ft = C | D | N | V | A | P | Wh 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 (Set (SO)) | O PhTree deriving (Show, Eq, Ord)
type LSO = (SO, Label)
type WS = Set (LSO)

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

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

-- constituents with no remembered features can be 'forgotten'
t :: [LSO] -> [LSO]
t = filter (\lso -> snd lso /= ([],[]))

-- return number of nodes in SO (here we can disregard multidominance)
soSize :: SO -> Int
soSize (S so) = foldr (\x y -> (soSize x) + y) 1 (Set.toList so)
soSize (O (Ps ts)) = foldr (\x y -> (soSize (O x)) + y) 1 ts
soSize _ = 1

-- given LSOs from a positive WS, return largest, i.e. the head
maxx :: [LSO] -> LSO
maxx lsos = foldr1 (\x y -> if ((soSize.fst) x) >= ((soSize.fst) y) then x else y) lsos

-- given WS, return matching ([head,comp], [other LSOs])
match:: [WS] -> ([LSO],[LSO])
match (ws0:wss) = case List.partition ((/= []).fst.snd) (Set.toList ws0) of
    ([h], others) -> let f = ((head.fst.snd) h) in case (List.partition ((== f).head.snd.snd) others, wss) of
      (([c], others'), []) -> ([h,c], others')
      (([],_),[ws1]) -> case List.partition ((== f).head.snd.snd) (Set.toList ws1) of
        ([c], others') -> ([h,c], others ++ others')

-- if LSOs satisfy shortest move constraint, return WS
smc :: [LSO] -> WS
smc lsos = if smc' [] lsos then (Set.fromList lsos) else (error "smc violation")
  where
    smc' _ [] = True
    smc' sofar ((s,([],p:ps)): lsos) = if elem p sofar then False else smc' (p:sofar) lsos
    smc' sofar (_:lsos) = smc' sofar lsos

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

A.2. Haskell: Derivations with unbounded merge

Replacing Data.Set with Data.MultiSet, and extending match and ck for Kleene-plus features.

```
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
import qualified Data.List as List

data F = C | D | N | V | A | P | Wh | Pred | Predx | T | K | Vx | Scr |
         Modal | Have | Be | Been | Ving | Ven deriving (Show, Eq, Ord)
data Ft = One F | Plus F deriving (Show, Eq, Ord) -- build features from base F
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 LSO = (SO, Label)
type WS = MultiSet (LSO)

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

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

t :: [LSO] -> [LSO]     -- constituents with no remembered features can be 'forgotten'
t = filter (\lso -> snd lso /= ([],[]))

soSize :: SO -> Int
soSize (S so) = foldr (\x y -> (soSize x) + y) 1 (MultiSet.toList so)
soSize (O (Ps ts)) = foldr (\x y -> (soSize (O x)) + y) 1 ts
soSize _ = 1

maxx :: [LSO] -> LSO
maxx lsos = foldr1 (\x y -> if ((soSize.fst) x) >= ((soSize.fst) y) then x else y) lsos

match:: [WS] -> ([LSO],[LSO])
match (ws0:wss) = case List.partition ((/= []).fst.snd) (MultiSet.toList ws0) of
    ([h], others) -> let (f,plus) = ((fplus.head.fst.snd) h) in
      case (List.partition ((== (One f)).head.snd.snd) others,wss) of
        (([c], others'),[]) -> ([h,c], others')  -- IM for One feature; otherwise EM:
        (([],_),ws1:wss) -> let lsos = (MultiSet.toList ws1) in let c = maxx lsos in
          if (((/= One f).head.snd.snd) c)
          then (error "match complement feature clash")
          else let others' = List.filter (/= c) lsos in
            if plus && others' == others
            then (h:c:(atb (One f) others wss), others)
            else if wss == [] then ([h,c], others ++ others') else (error ("match: too many wss"))
  where
    fplus :: Ft -> (F, Bool)           -- parse the feature
    fplus ft = case ft of (One f) -> (f, False); (Plus f) -> (f, True)

    atb :: Ft -> [LSO] -> [WS] -> [LSO] -- collect comps with first feature f and others=movers
    atb _ _ [] = []
    atb ft movers (ws:wss) = case List.partition ((== ft).head.snd.snd) (MultiSet.toList ws) of
      ([c'], others) -> if others == movers then c':(atb ft movers wss) else (error "match: ATB error")

smc :: [LSO] -> WS
smc lsos = if smc' [] lsos then (MultiSet.fromList lsos) else (error "smc violation")
  where
    smc' _ [] = True
    smc' sofar ((s,([],p:ps)):lsos) = if elem p sofar then False else smc' (p:sofar) lsos
    smc' sofar (_:lsos) = smc' sofar lsos

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

A.3. Haskell: Derivation as transduction

```haskell
module MgTransduction 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
import qualified Data.List as List

import Mg

negWS :: WS -> Bool
negWS ws = fst (MultiSet.partition ((/= []).fst.snd) ws) /= (MultiSet.empty)

ell :: SO -> WS
ell (L (w,l)) = (MultiSet.singleton (L (w,l), l))
ell (S s) = case List.partition negWS (map ell (MultiSet.toList s)) of
  ([nws],pws:pwss) -> case List.partition ((/= []).fst.snd) (MultiSet.toList nws) of
      ([(nso,(f:ns,ps))], others) -> case List.filter ((== f).head.snd.snd) others of
          [(pso, plabel)] -> if (fst.maxx) (MultiSet.toList pws) /= pso  -- IM
                             then (error "ell: move-over-merge violation")
                             else d [nws]
          [] -> d (nws:pws:pwss) -- EM
```

A.4. Haskell: Head movement

```haskell
module MgHm 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
import qualified Data.List as List

import Mg
import MgTransduction
import MgLinearization

-- where i = #heads needed by selector above, (h i so) = (heads, so')
h :: Int -> SO -> ([String], SO)
h 0 (L lex) = ([], L lex)
h i (S s) = case List.partition negWS (map ell (MultiSet.toList s)) of
  ([nws],pws:pwss) ->
    case (fst.maxx.(MultiSet.toList)) nws of
      L (w,fs) -> let i' = inc w in
          let (hs,pso) = h i' ((fst.maxx.(MultiSet.toList)) pws) in
          let psos = atbh i' hs pwss in
            if i == 0
            then ([], S (MultiSet.fromList (L (hs ++ w, fs) : pso : psos)))
            else (hs ++ w, S (MultiSet.fromList (L ([], fs) : pso : psos)))
      nso ->
          let (hs,nso') = h i nso in
          let psos = (map (fst.maxx.(MultiSet.toList)) (pws:pwss)) in
            (hs, S (MultiSet.fromList (nso' : psos)))

atbh :: Int -> [String] -> [WS] -> [SO] -- collect comps with hs extracted
atbh _ _ [] = []
atbh i hs (pws:pwss) = case h i ((fst.maxx.(MultiSet.toList)) pws) of
      (hs', pso) -> if hs' == hs then pso:(atbh i hs pwss) else (error "atbh error")

inc :: [String] -> Int -- map [w] to # head-incorporator +'s on w, else 0
inc [] = 0
inc (w:_) = cc w where
    cc ('+' : s) = 1 + (cc s)
    cc _ = 0
```

A.5. Haskell: Linearization

```
module MgLinearization 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
import qualified Data.List as List

import Mg
import MgTransduction

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 plsos = (map (maxx.(MultiSet.toList)) pwss) in
           let ts = map (\x -> case (o_svo.fst) x of (O t) -> t) plsos in
             case (o_svo nso, o_svo pso, posfs) of
               (O (Pl i),O t,_:_:_) -> (O ((Pl i)))
               (O (Pl i),O t,_) -> (O (Ps ((Pl i):t:ts)))
               (O t,O t',_:_:_) -> (O 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 plsos = (map (maxx.(MultiSet.toList)) pwss) in
           let ts = map (\x -> case (o_sov.fst) x of (O t) -> t) plsos in
            case (o_sov nso, o_sov pso, posfs) of
              (O t,O t',_:_:_) -> (O t)
              (O t,O t',_) -> (O (Ps ((t':ts)++[t])))

-- map SO to what oering usually depends on: (head, comp, head_features, comp_features, otherCompWSs)
o :: SO -> (SO,SO,[Ft],[Ft],[WS])
o (S s) = case List.partition negWS (map ell (MultiSet.toList s)) of  -- NB! inefficient
  ([nws],pws:pwss) -> case List.partition ((/= []).fst.snd) (MultiSet.toList nws) of
     ([(nso,(f:ns,ps))], others) -> case ((List.filter ((== f).head.snd.snd) others),pwss) of
        ([(pso,([],posfs))], []) -> (nso, pso, ps, posfs, pwss)       -- for IM, else EM:
        ([],_) -> let (pso,([],posfs)) = (maxx.(MultiSet.toList)) pws in (nso, pso, ps, posfs, pwss)
```

B Implementations: Python

Implementing minimalist grammars in Python is slightly harder than in Haskell. Keeping the two implementations similar, this Python code is written in a functional style, and so 'type hints' are added to allow each file to be checked with mypy after any changes.

See https://github.com/epstabler/mgt for this code and examples that use these functions, with command-line and nltk-based graphical display.

B.0. 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 files mgBinTests.py and mgTests.py have 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 mgBinTests 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)

>>>
```

We will use these common list functions

```
""" listfs.py
    It is convenient to have these two list functions
    Since written in a functional style, annotated for mypy.
"""

def partition(f, lst:list) -> tuple:
    """ given boolean function f, partition list into two lists: (f-elements, non-f elements) """
    yes, no = [], []
    for d in lst:
        if f(d): yes.append(d)
        else: no.append(d)
    return (yes, no)

def unzip(pairs: list) -> tuple:
    """ we could use zip(*pairs), but mypi typing does not understand """
    firsts, seconds = [], []
    for (x,y) in pairs:
        firsts.append(x)
        seconds.append(y)
    return (firsts, seconds)
```

Python also does not allow sets of sets. So we use frozensets as our sets, and define these functions:

```
""" fset.py (f is for "frozen", i.e. immutable)
Since python sets cannot contain sets, we use frozensets.
>>> x = set([1,2])
>>> y = set([3,4])
>>> z = set([x,y])
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'set'

>>> import fset
>>> x = fset.fromList([1,2])
>>> y = fset.fromList([3,4])
>>> z = fset.fromList([x,y])

since written in a functional style, annotated for mypy.
"""

def fromList(lst:list) -> frozenset:
    """ given list, return fset (i.e. frozenset) of its elements """
    return frozenset(lst)

def toList(fs:frozenset) -> list:
    """ given fset (i.e. frozenset), return its elements in a list """
    return list(fs)

def partition(f,fs:frozenset) -> tuple:
    """ given boolean function f and fset (i.e. frozenset), return (f-elements,non-f-elements) """
    yes, no = [], []
    for d in fs:
        if f(d): yes.append(d)
        else: no.append(d)
    return (frozenset(yes), frozenset(no))
```

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’. Then, the needed multiset functions can be defined like this:

```
""" fmultiset.py (f is for "frozen", i.e. immutable)
Since python multisets cannot contain multisets, we use frozendicts.

NB: mypy does not know the datatype frozendict, so no type hints here.
"""
import frozendict

def fromList(lst):
    """ given list, return multiset (i.e. frozendict) of those elements """
    counts = {}
    for e in lst:
        if not(e in counts.keys()): counts[e] = 1
        else: counts[e] += 1
    return frozendict.frozendict(counts)

def toList(fms):
    """ given fmultiset (i.e. frozendict), return its elements in a list """
    lst = []
    for y in fms.keys():
        lst.extend(fms[y] * [y])
    return lst

def partition(f,fms):
    """ given boolean function f and fmultiset (i.e. frozendict), return (f-elements,non-f-elements) """
    yes, no = {}, {}
    for d in fms.keys():
        if f(d): yes[d] = fms[d]
        else: no[d] = fms[d]
    return (frozendict.frozendict(yes), fd.frozendict(no))
```

B.1. Python: Derivations with binary merge

```
""" mgBin.py (since written in a functional style , annotated for mypy) """
import fset
from listfs import * # for partition , unzip

def mrg(lst: list) -> frozenset:
  """ merge """
  return fset.fromList( lst )

def ck(labels: list) -> list:
  """ remove first features from already matched labels [0] and labels [1] """
  [(nns,nps), ((),pps)] = labels
  return [(nns[1:],nps), ((),pps[1:])]

def t(lsos:list) -> list:
  """ remove LSOs with no features in their label """
  return [x for x in lsos if x[1][1]]

def soSize(so) -> int:
  """ calculate the size of a syntactic object """
  if isinstance(so,tuple) and len(so) == 2 and \
     isinstance(so[0],tuple) and ( len(so[0]) == 0 or isinstance(so[0][0],str) ): # lex
    return 1
  elif isinstance(so,frozenset): # fset
    return 1 + sum(map(soSize, fset.toList(so)))
  else: # phtree
    return 1 + sum(map(soSize, so))

def maxx(lsos: list) -> tuple:
  """ given LSOs (of a WS), return LSO with largest SO (the head) """
  sofar = 0
  maxLSO = (((),((),()))
  for lso in lsos:
    lsoSize = soSize(lso[0])
    if lsoSize > sofar:
      sofar = lsoSize
      maxLSO = lso
  return maxLSO

def smc(lsos: list) -> frozenset:
  """ if LSOs have no pos feature in common, return them """
  firstPosFeatures = [lso[1][1][0] for lso in lsos if lso[1][0] == ()]
  if len(firstPosFeatures) < len(set(firstPosFeatures)):
    return fset.fromList([])
  else:
    return fset.fromList(lsos)

def match(wss:list) -> tuple:
  """ given list of workspaces, return ([head,complement],otherLSOs) """
  (ws0,wss0) = (wss[0],wss[1:])
  (heads, others) = partition (lambda lso: lso[1][0] != (), fset.toList(ws0))
  if not(len(heads)==1):
    raise RuntimeError('match: 0 or >1 neg lsos in ws0')
  h = heads[0]
  f = h[1][0][0]
  (ics, iothers) = partition (lambda lso: lso[1][1][0] == f, others)
  if ics and wss0 == []:   # im
    return ([h,ics[0]], iothers)
  elif ics==[] and len(wss0)==1: # em
    ws1 = wss0[0]
    lsos = fset.toList(ws1)
    c = maxx(lsos)
    if c[1][0] == () and c[1][1][0] == f:
      others1 = [x for x in lsos if x != c]
      return ([h,c], others + others1)
    else: RuntimeError('match: complement')
  else:
    raise RuntimeError('match')

def d(wss:list) -> frozenset:
  """ the derivational step: given list of workspaces, return derived workspace """
  (matched, movers) = match(wss)
  (sos, labels) = unzip(matched)
  return smc( t( list(zip( [mrg(sos)]+sos[1:], ck( labels ))) + movers))
```

B.2. Python: Derivations with unbounded merge

Replacing fset.py (i.e. frozensets) with fmultiset.py (i.e. frozendicts), and extending match, ck.

```
""" mg.py (not annotated for mypy, since mypy does not know frozendict ) """
import frozendict, fmultiset
from listfs import * # for partition , unzip

def mrg(lst: list):
  """ merge """
  return fmultiset.fromList( lst )

def ck(labels: list) -> list:
  """ remove first features from already matched labels [0], labels [1] and T for rest """
  (nns,nps ),( () ,pps) = labels [0], labels [1]
  return ((nns[1:],nps ), ( () ,pps[1:])) + tuple([( () , () ) for label in labels[2:]])

def t( lsos : list) -> list:
  """ remove LSOs with no features in their label """
  return [x for x in lsos if x[1][1]]

def soSize(so) -> int:
  """ calculate the size of a syntactic object """
  if isinstance(so,tuple) and len(so) == 2 and \
     isinstance(so[0],tuple) and ( len(so[0]) == 0 or isinstance(so[0][0],str) ): # lex
    return 1
  elif isinstance(so,frozendict.frozendict): # fmultiset
    return 1 + sum(map(soSize, fmultiset. toList (so )))
  else: # phtree
    return 1 + sum(map(soSize, so))

def maxx(lsos: list) -> tuple:
  """ given LSOs (of a WS), return LSO with largest SO (the head) """
  sofar = 0
  maxLSO = (( () ),( () , () ))
  for lso in lsos:
    lsoSize = soSize( lso [0])
    if lsoSize > sofar :
      sofar = lsoSize
      maxLSO = lso
  return maxLSO

def smc(lsos: list):
  """ if LSOs have no pos feature in common, return them """
  firstPosFeatures = [ lso [1][1][0] for lso in lsos if lso [1][0] == () ]
  if len( firstPosFeatures ) < len(set(firstPosFeatures )):
    return fmultiset.fromList( [] )
  else:
    return fmultiset.fromList( lsos )

def match(wss:list) -> tuple:
  """ given list of workspaces, return ([head,complement],otherLSOs) """
  (ws0,wss0) = (wss [0], wss [1:])
  (heads, others ) = partition (lambda lso: lso [1][0] != () , fmultiset. toList (ws0))
  if not(len(heads)==1): raise RuntimeError("match: 0 or >1 neg lsos in ws0")
  h = heads [0]
  f = h [1][0][0]
  if f[-1] == '+':
    f, plus = f [:-1], True
  else:
    plus = False
  ( ics , iothers ) = partition (lambda lso: lso [1][1][0] == f, others )
  if ics and wss0 == [] :   # im
    return ([h,ics [0]], iothers )
  elif ics==[] and len(wss0)>0: # em
    ws1 = wss0[0]
    lsos = fmultiset . toList (ws1)
    c = maxx(lsos)
    if c [1][1][0] != f:
      raise RuntimeError("match: %r != %r" % (f,c[1][1][0]))
    else:
      others1 = [x for x in lsos if x != c]
      if plus and others == others1 :
        return ([h,c]+atb(f, others , wss0 [1:]), others )
      elif wss0[1:] == [] :
        return ([h,c], others + others1 )
      else:
        raise RuntimeError("match: too many wss")

def atb(f, movers, wss):
  """ collect comps with first feature f and others==movers """
  if wss == [] :
    return []
  else:
    ws1 = wss[0]
    lsos = fmultiset . toList (ws1)
    c = maxx(lsos)
```

```
    if c[1][1][0] != f:
      raise RuntimeError("match: complement clash")
    else:
      others = [x for x in lsos if x != c]
      if others != movers:
        raise RuntimeError("match: mover clash")
      else:
        return [c] + atb(f, movers, wss[1:])

def d(wss:list):
  """ the derivational step: given list of workspaces, return derived workspace """
  (matched, movers) = match(wss)
  (sos, labels) = unzip(matched)
  return smc( t( list(zip( [mrg(sos)]+sos[1:], ck( labels ))) + movers))
```

B.3. Python: Derivation as transduction

```
""" mgTransduction.py (not annotated for mypy, since mypy does not know frozendict) """
import frozendict
from mg import *

def negWS(ws:list) -> bool:
  return [lso for lso in fmultiset.toList(ws) if lso[1][0] != () ] != []

def ell(so):
  """ map so to its derived workspace """
  if isinstance(so,tuple) and len(so) == 2 and \
     isinstance(so[0],tuple) and ( len(so[0]) == 0 or isinstance(so[0][0],str) ):
    return fmultiset.fromList([(so,so[1])])
  elif isinstance(so,frozendict.frozendict):
    ([nws],pwss) = partition (negWS, [ell(s) for s in fmultiset.toList(so)])
    pws = pwss[0]
    (nlsos, others) = partition ( lambda lso: lso[1][0] != (), fmultiset.toList(nws) )
    if len(nlsos) != 1:
      raise RuntimeError("ell: zero or >1 neg LSOs in workspaces")
    (nso,(nfs,pfs)) = nlsos[0]
    imatches = [lso for lso in others if lso[1][1][0] == nfs[0]]
    if len(imatches)==1:
      (pso, plabel) = imatches[0]
      if maxx(fmultiset.toList(pws))[0] != pso or pwss[1:] != []:
        raise RuntimeError("move-over-merge error")
      else: # im
        return d([nws])
    else: # em
      return d([nws] + pwss)
  else:
    raise TypeError("ell type error")
```

B.4. Interfaces in Python: Head movement

** under revision **

B.5. Interfaces in Python: Linearization

```python
""" mgLinearization.py (not annotated for mypy, since mypy does not know frozendict ) """
import frozendict
from mgTransduction import *

def ord_svo(so) -> tuple:
  """ maps so to ordered phTree """
  if isinstance(so,tuple) and len(so) == 2 and \
     isinstance(so[0],tuple) and ( len(so[0]) == 0 or isinstance(so[0][0],str) ): # so is lex
    return so
  else: # so is mset
    (nso, pso, _, posfs, pwss) = ord(so)
    nt, pt = ord_svo(nso), ord_svo(pso)
    plsos = list(map(maxx, pwss))
    psos = [x[0] for x in plsos]
    pts = map(ord_svo, psos)
    if isinstance(nt,tuple) and len(nt) == 2 and \
       isinstance(nt[0],tuple) and ( len(nt[0]) == 0 or isinstance(nt[0][0],str) ): # nt is lex
      if len(posfs) > 1: return (nt,)
      else: return (nt, pt) + tuple(pts)
    else: # nt is mset
      if len(posfs) > 1: return (nt,)
      else: return (pt,) + tuple(pts) + (nt ,)

def ord_sov(so) -> tuple:
  """ maps so to ordered phTree """
  if isinstance(so,tuple) and len(so) == 2 and \
     isinstance(so[0],tuple) and ( len(so[0]) == 0 or isinstance(so[0][0],str) ): # so is lex
    return so
  else: # so is mset
    (nso, pso, _, posfs, pwss) = ord(so)
    nt, pt = ord_sov(nso), ord_sov(pso)
    plsos = list(map(maxx, pwss))
    psos = [x[0] for x in plsos]
    pts = map(ord_sov, psos)
    if len(posfs) > 1: return (nt,)
    else: return (pt,) + tuple(pts) + (nt ,)

def ord(so) -> tuple:
  """ maps so to (head, comp, head_pos_features, comp_positive-features, otherPosWSs) """
  if not(isinstance(so,frozendict.frozendict)): raise TypeError("ord type error")
  ([nws],pwss) = partition (negWS, [ell(s) for s in fmultiset.toList(so)]) # NB! inefficient
  (pws,pwss1) = (pwss[0],pwss[1:])
  (nlsos, others) = partition ( lambda lso: lso[1][0] != (), fmultiset.toList(nws) )
  (nso,(fns,ps)) = nlsos[0]
  f = fns[0]
  imatches = [lso for lso in others if lso[1][1][0] == f]
  if len(imatches)==1: # im
    (pso,([],posfs)) = imatches[0]
    return (nso, pso, ps, posfs, pwss1)
  else: # em
    (pso,([],posfs)) = maxx(fmultiset.toList(pws))
    return (nso, pso, ps, posfs, pwss1)
```