# 0 Basic types

```haskell
module MgTypes where

type Morph = [String]                                    -- E.g. initially ["√cat"], then phon specified ["cat"]
type F = String                                          -- feature -- E.g. "N", "V", "ϕ:3p", "-ϕ:_", "-ϕ:_3p"
type Sel = ([F], [F])                                    -- (neg features -o pos features)
type Agr = [([[F]], F)]                                  -- list of (tier, feature) pairs
type Leaf = (Morph, Sel, Agr)                            -- basic syntactic objects
data SO = L Leaf | S SO SO | E deriving (Show, Eq, Ord)  -- syntactic objects (E is for SO deleted by lin)
type Mover = (SO, [F], Agr)                              -- (movingSO, positive sel features, agr features)
type Label = (Sel, Agr, [Mover])                         -- label computed by sel

fstOf3 (x,_,_) = x
sndOf3 (_,x,_) = x
thrOf3 (_,_,x) = x
```

# 1 Minimalist grammar, composed

```haskell
module Mg where
import MgTypes (SO)
import Mrg (mrg)
import Sel (sel)
import Agr (agr)
import Hm (hm)
import Lin (lin)
import Vi (vi)

g :: SO -> SO
g = vi . lin . hm . agr . sel . mrg               -- g is the composition of 6 functions
```

# 2 Merge

```haskell
module Mrg where
import MgTypes (SO)

mrg :: SO -> SO
mrg = id         -- Haskell *already enforces* the binary SO structure of MgTypes.hs, so this suffices
```

# 3 Selection

```haskell
module Sel where
import Data.List (partition)
import MgTypes (SO(L,S),Label,sndOf3)

sel :: SO -> SO
sel so = let lso = (lbl so, lbl so) in
  if lso == lso then so else error "ls"      -- Haskell is lazy. Here == forces computation of lso

lbl :: SO -> Label
lbl (L (_, (negs,poss), agrfs)) = ((negs, poss), agrfs, [])
lbl (S so so') = case (lbl so, lbl so') of
  ( ((f:ns, ps), agrfs, movers), (([], g:ps'), agrfs', movers') ) -> case partition ((== f).head.sndOf3) movers of
      ( [(y',_:ps'',agrfs'')], movers'') ->         -- internal merge
         if y' == so' then ((ns,ps), agrfs, newmovers so' movers'' [] ps'' agrfs'') else error "move-over-merge"
      _ ->                                          -- external merge
         if f == g then ((ns,ps), agrfs, newmovers so' movers movers' ps' agrfs') else error "unlabelable, no match"
  _ -> error "unlabelable"
  where
  newmovers so' ms ms' ps' agrfs' = let nms = case ps' of {[] -> ms++ms'; _ -> ms++[(so',ps',agrfs')]++ms'} in
    if smc nms then nms else error "smc"
  -- smc _ = True                                   -- uncomment this line to remove smc check
  smc movers = noduplicates (map (head.sndOf3) movers)
  noduplicates fs = case fs of {[] -> True; f:fs -> f `notElem` fs && noduplicates fs}
```

# 4 Agreement

```haskell
module Agr where
import Data.List (partition,union)
import MgTypes (F,Agr,Leaf,SO(L,S,E),Label,fstOf3,sndOf3)

{- In development ... this interim version not working as intended yet -}

agr :: SO -> SO
agr so = let isos = instantiate so in if isos == isos then so else error "agr"  --  == forces computation of isos

instantiate so = let (iso, ispine) = i (unionTiers so) [] so in iso  -- return fully instantiated agreement
  where
  i tier ders so = case updateSpine tier (hd so) ders of -- i tier dcommanders so -> (so',dcommandees)
    Just spine -> let (so',dees) = i' tier spine so in iLeaf ders dees so'
    Nothing -> i' tier ders so

  i' tier ders (L leaf) = (L leaf,[])  -- i' tier dcommanders so -> (so',dcommandees)
  i' tier ders (S so so') = case (lblfs so, lblfs so') of  -- headFirst?
    ( ((f:ns, ps), agrfs, movers), (([], _:ps'), agrfs', movers') ) -> i'' f  so movers so' tier ders True
    ( (([], _:ps'), agrfs', movers'), ((f:ns, ps), agrfs, movers) ) -> i'' f so' movers so  tier ders False

  i'' f so movers so' tier spine headFirst = case partition ((== f).head.sndOf3) movers of -- IM or EM?
    ( [(y',_:ps'',agrfs'')], movers'') ->        -- IM
      let (iso,ispine) = i tier [] so  in if headFirst then (S iso so',ispine) else (S so' iso,ispine)
    _ -> case updateSpine tier (hd so') spine of -- EM
      Just spine' -> let (iso,ispine) = i tier spine' so in let (iso', _) = i tier [] so' in
        if headFirst then (S iso iso', ispine) else (S iso' iso, ispine)
      Nothing -> let (iso,ispine) = i tier spine so in let (iso', _) = i tier [] so' in
        if headFirst then (S iso iso', ispine) else (S iso' iso, ispine)

iLeaf ders dees so' = let (r,sel,agr) = hd so' in -- iLeaf dcommanders dcommandees so -> (so',dcommandees')
  let (so'',dees',agr') = iAgr ders (so',dees,agr) in (putHd (r,sel,agr') so'',dees')

iAgr _ (so,dees,[]) = (so,dees,[])
iAgr ders (so,dees,(tier,f'):agr) = let (so',dees',f'') = iF ders (so,dees) tier f' in
  let (so'',dees'',agr') = iAgr ders (so',dees',agr) in (so'',dees'',(tier,f''):agr')

iF ders (so,dees) tier f' = case nextInTier tier dees of
  Just (r,sel,agr) -> let (t,v) = typeVal f' in let (agr',so',f'') = matchAgr t v agr so in
      if head v == '_' then (putTier tier (r,sel,agr') so', dees, f'') else (so,(r,sel,agr'):dees,f'')
  Nothing -> case nextInTier tier ders of
    Just (r,sel,agr) -> let (t,v) = typeVal f' in let (agr',so',f'') = matchAgr t v agr so in
        if head v == '_' then (putTier tier (r,sel,agr') so', dees, f'') else (so,(r,sel,agr'):dees,f'')
    Nothing -> (so,dees,f')

matchAgr t v [] so = ([],so,t++":"++v)
matchAgr t v ((tier,f):agr) so = let (t',v') = typeVal f in if t == t'
  then let x = matchF v v' in ((tier,t++":"++x):agr,so,t++":"++x)
  else let (agr',so',f') = matchAgr t v agr so in ((tier,f):agr',so',f')

matchF ('_':x) x' = case x of {"" -> x'; _ -> if x == x' then x else error "matchF downward conflict"}
matchF x' ('_':x) = case x of {"" -> x'; _ -> if x == x' then x else error "matchF upward conflict"}
matchF x x' = if x == x' then x else error "matchF"

hd (L leaf) = leaf
hd (S so so') = if (not.null.fst.fstOf3.lblfs) so then hd so else hd so'

putHd leaf (L _) = L leaf
putHd leaf (S so so') = if (not.null.fst.fstOf3.lblfs) so then S (putHd leaf so) so' else S so (putHd leaf so')

putTier _ leaf (L _) = L leaf
putTier tier leaf (S so so')
 | leafInTier (hd (S so so')) tier = putHd leaf (S so so')
 | (not.null.fst.fstOf3.lblfs) so = S (putTier tier leaf so) so'
 | otherwise = S so (putTier tier leaf so')

updateSpine tier leaf spine = if leafInTier leaf tier then Just (leaf:spine) else Nothing

nextInTier tier spine = case spine of {[] -> Nothing; lf:spine -> if leafInTier lf tier then Just lf else nextInTier tier spine}

leafInTier (_,(_,p:ps),agr) = foldr (\x sofar -> subList x (p:map snd agr) || sofar) False

typeVal f = tV' "" f where tV' pre f = case f of {"" -> (pre,""); (c:cs) -> if c == ':' then (pre,cs) else tV' (pre++[c]) cs}
```

```
unionTiers (S so so') = unionTiers so `union` unionTiers so' -- return union of tiers and probe host categories
unionTiers (L (_,sel,agr)) = foldr (union.fst) [[(head.snd) sel] | hasProbe agr] agr

hasProbe = foldr (\x sofar -> ((== '_').head.snd.typeVal.snd) x || sofar) False -- agr has a probe?

subList a b = foldr (\x sofar -> (x `elem` b) && sofar) True a  -- a is a sublist of b?

lblfs (L (_, (ns,ps), agr)) = ((ns,ps), agr, []) -- since lbl already checked by sel, this is simpler: no smc, etc
lblfs (S so so') = case (lblfs so, lblfs so') of -- head-first?
  ( ((f:ns, ps), agr, movers), (([], _:ps'), agr', movers') ) -> lblfs' so f (ns,ps) agr movers so' ps' agr' movers'
  ( (([], g:ps'), agr', movers'), ((f:ns, ps), agr, movers) ) -> lblfs' so' f (ns,ps) agr movers so  ps' agr' movers'
  where
  lblfs' so f headFs agr movers spec ps' agr' movers' =  case partition ((== f).head.sndOf3) movers of -- IM or EM?
    ( [(y',_:ps'',agr'')], movers'') ->  (headFs, agr, newmoversfs spec movers'' [] ps'' agr'')
    _ -> (headFs, agr, newmoversfs spec movers movers' ps' agr')
  newmoversfs spec mover mover' ps' agr' = case ps' of { [] -> mover++mover'; _ -> mover++[(spec,ps',agr')]++mover' }

agrees f agr = case agr of {[] -> False; ((_,f'):agr) -> f == f' || agrees f agr} -- is f in agr?
```

# 5 Head movement

```
module Hm where
import MgTypes (SO(L,S),Morph)

hm :: SO -> SO
hm (L leaf) = L leaf
hm so = case h False False [] so of { ([], so') -> so' ; (rs,so') -> error ("hm: " ++ show rs) }
  where
  dependent morph = not (null morph)  &&  (head.head) morph == '-'
  strong morph = not (null morph) && (last.last) morph == '$'
  -- h dep-above? strong-above? heads-from-above input-SO -> (span-heads-still-to-be-placed, output-SO)
  h :: Bool -> Bool -> Morph -> SO -> (Morph, SO)
  h hiDep hiStrong rs (L (r,sel,agr)) =
    if hiDep && not (strong r && not hiStrong)
    then (r++rs, L ([],sel,agr))
    else ([], L (r++rs,sel,agr))
  h hiDep hiStrong rs (S (L (r,sel,agr)) so) =             -- note: relies on head-first, must precede lin
    if dependent r
    then let (rs',so') = h True (hiStrong || strong r) (r++rs) so in
      if not hiDep || (strong r && not hiStrong)
      then ([], S (L (rs',sel,agr)) so')
      else (rs', S (L ([],sel,agr)) so')
    else case h False False [] so of
      ([], so' ) ->
        if hiDep
        then (r++rs, S (L ([],sel,agr)) so')
        else ([], S (L (r++rs,sel,agr)) so')
      (rs'', _ ) -> error ("h: not dep but heads moved from below: " ++ show rs'')
  h hiDep hiStrong rs (S so so') =
    let (rs',so'') = h hiDep hiStrong rs so in (rs', S so'' so')
```

# 6 Linearization

```
module Lin where
import MgTypes (SO(L,S,E),fstOf3,sndOf3)
import Data.List (partition)
import Sel (lbl)

lin :: SO -> SO
lin (L leaf) = L leaf
lin (S so so') = let ((f:_,ps),agrfs,movers) = lbl so in case partition ((== f).head.sndOf3) movers of
    ( [(_,ps'',_)], movers'' ) -> ord ps'' so so'  -- internal merge
    _ -> ord ((snd.fstOf3.lbl) so') so so'         -- external merge
  where
    ord posFeats so so' = case (posFeats, so) of
      ([_], L _) -> S (lin so) (lin so')      -- first merge, nonmoving complement
      ([_], S _ _) -> S (lin so') (lin so)    -- nonfirst merge, nonmoving complement
      _ -> S (lin so) (del so')               -- moving element
    -- del _ = E                              -- uncomment this line to make del = removal
    del (S so so') = S (del so) (del so')
    del (L (morph,sel,agr)) = if head morph == "(" then L (morph,sel,agr) else L (["("]++morph++[")"],sel,agr)
```

# 7 Vocabulary insertion

```haskell
module Vi where
import MgTypes (Morph,Sel,Agr,SO(L,S,E))
import Agr (instantiate,agrees)

{- In development ... Vi rules integrated into function definitions here, facilitating experimentation.
 They should be separated in future work, to support parametric grammar definition, as in the paper. -}

vi :: SO -> SO        -- vocabulary insertion
vi so = vi' (instantiate so)
  where
  vi' E = E
  vi' (S so so') = S (vi' so) (vi' so')
  vi' (L ([],sel,agr)) = L ([],sel,agr)
  vi' (L (r:rs,sel,agr)) =
    if strip r `elem` ["√pres","√past"]
    then L ("DO":viDefault (strip r:rs) sel agr,sel,agr)
    else L (viDefault (strip r:rs) sel agr, sel, agr)
  strip ('-':cs) = stripSuf cs             -- remove morph diacritics
  strip cs = stripSuf cs
  stripSuf r = if not (null r) && last r == '$' then init r else r

  viDefault :: Morph -> Sel -> Agr -> Morph
  viDefault rs sel agr = case rs of
    ("-√pres":rs') -> if "ϕ:3s" `agrees` agr then "-s":viDefault rs' sel agr else viDefault rs' sel agr
    ("-√past":rs') -> "-ed":viDefault rs' sel agr
    ("√decl":rs') -> viDefault rs' sel agr        -- unpronounced
    ("√q":rs') -> viDefault rs' sel agr           -- unpronounced
    ("√wh":rs') -> viDefault rs' sel agr          -- unpronounced
    ("-√v$":rs') -> viDefault rs' sel agr         -- unpronounced
    (r:rs') ->
      if "N" `elem` snd sel && "ϕ:3p" `agrees` agr
      then defStrip r:"-s":viDefault rs' sel agr
      else defStrip r:viDefault rs' sel agr
    [] -> []

  defStrip ('√':cs) = cs  -- use root name as default
  defStrip cs = cs        -- else id

ph :: SO -> String  -- return string of phonologically specified morphs
ph (S so so') = joinstr " " [ph so, ph  so']
ph (L (w,_,_)) = if not (null w) && head w == "(" then "" else joinstr " " w
ph E = ""

joinstr :: String -> [String] -> String  -- join strings, separated by sep
joinstr sep = foldr (\x y -> if null y then x else x ++ sep ++ y) ""
```

# Example session with ghci

Note that lin puts (moved) (constituents) in parentheses, rather than deleting them,
just to make it easier to understand the analyses.
To delete them, uncomment the line that defines this in Lin.hs.
Smc is enforced in Sel.hs. To stop enforcing it, adjust definition of newmovers.
Here, ex 8 is the example from Figures 1 and 4 in the paper. See Examples.hs for many more.

```
$ ghci -package time
Loaded package environment from /Users/es/.ghc/aarch64-darwin-9.6.7/environments/default
GHCi, version 9.6.7: https://www.haskell.org/ghc/  :? for help
ghci> :l Examples
[ 1 of 11] Compiling MgTypes          ( MgTypes.hs, interpreted )
[ 2 of 11] Compiling Hm               ( Hm.hs, interpreted )
[ 3 of 11] Compiling ExampleAtoms     ( ExampleAtoms.hs, interpreted )
[ 4 of 11] Compiling Agr              ( Agr.hs, interpreted )
[ 5 of 11] Compiling Mrg              ( Mrg.hs, interpreted )
[ 6 of 11] Compiling Sel              ( Sel.hs, interpreted )
[ 7 of 11] Compiling Lin              ( Lin.hs, interpreted )
[ 8 of 11] Compiling Vi               ( Vi.hs, interpreted )
[ 9 of 11] Compiling PrettyPrint      ( PrettyPrint.hs, interpreted )
[10 of 11] Compiling Mg               ( Mg.hs, interpreted )
[11 of 11] Compiling Examples         ( Examples.hs, interpreted )
Ok, 11 modules loaded.

ghci> ex 8

input so =
{ √decl:V -o C,
 { { √know:C.D -o V,
     { { √q:V.wh -o C,
         { { √like:D.D -o V,
             { √which:N -o D.wh,
               √food:N } },
           { √the:N -o D,
             √cat:N } } },
       { √which:N -o D.wh,
         √food:N } } },
   √Jo:D } }

lbl so =
C

g so =
{ :V -o C,
 { Jo:D,
   { know:C.D -o V,
     { { which:N -o D.wh,
         food:N },
       { :V.wh -o C,
         { { the:N -o D,
             cat:N },
           { like:D.D -o V,
             { ( √which ):N -o D.wh,
               ( √food ):N } } } } } } }

execution time for computing (g so) and printing that result = 0.000918s

(ph.g) so =
 Jo know which food  the cat like

ghci>
```