0 Basic types

```
module MgTypes where

type Morph = [String]                                    -- E.g. initially ["√cat"], then phon specified ["cat"]
type F = String                                          -- Features. E.g. "N", "V", or (for Agr) "φ:_", "φ:3p", "κ:nom", "φ:_3p"
type Sel = ([F], [F])                                    -- (negative features, positive features)
type Goal = F                                            -- Agr Goal is a typed feature: "φ:3p", "φ:_", "φ:_3p"
type Tier = [[F]]                                        -- Tier is list of feature lists, e.g.: [["D"],["N"]], [["D","κ:nom"],["C"]]
type Probe = (Tier,F)                                    -- Probe (tier, typed feature) pair
type Agr = ([Probe],[Goal])                              -- Agr features are a list of probes and list of goals
type Leaf = (Morph, Sel, Agr)                            -- basic syntactic objects
data SO = L Leaf | S SO SO | E deriving (Show, Eq, Ord) -- hierarchical 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

typeVal :: F -> (F,F)  -- map agr feature to (type,value)
typeVal f = typeVal' ("",f) where typeVal' (t,f) = case f of {':':cs -> (t,cs); c:cs -> typeVal' (t++[c],cs); _ -> (t,"")}

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

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

joinStr separator = foldr (\x y -> if null y then x else x ++ separator ++ y) ""
```

1 Merge

```
module Mrg where
import MgTypes (SO)

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

2 Selection

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

sel :: SO -> SO
sel so = let lso = (lblck so, lblck so) in
  if lso == lso then so else error "ls"      -- Haskell is lazy. Here == forces computation of lso
  where
  lblck (L (_, selfs, agrfs)) = (selfs, agrfs, [])
  lblck (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'') ->        -- IM
            if y' == so' then ((ns,ps), agrfs, smc (newmovers so' movers'' [] ps'' agrfs'')) else error "move-over-merge"
        _ -> if f == g then ((ns,ps), agrfs, smc (newmovers so' movers movers' ps' agrfs')) else error "unlabelable, no match"
    _ -> error "unlabelable"
  smc movers = if noduplicates (map (head.sndOf3) movers) then movers else error "smc"
  noduplicates fs = case fs of {[] -> True; f:fs -> f `notElem` fs && noduplicates fs}

newmovers so' ms ms' ps' agrfs' = case ps' of {[] -> ms++ms'; _ -> ms++[(so',ps',agrfs')]++ms'}

hd so = case so of {S so _ -> hd so; L leaf -> leaf} -- after lablck and before lin, this def of syntactic head suffices

lbl (L (_, selfs, agrfs)) = (selfs, agrfs, []) -- after lblck, lbl can be calculated without identities or smc
lbl (S so so') = case (lbl so, lbl so') of
  ( ((f:ns, ps), agrfs, movers), (([], _:ps'), agrfs', movers') ) -> case partition ((== f).head.sndOf3) movers of
      ( [(_,_:ps'',agrfs'')], movers'') -> ((ns,ps), agrfs, newmovers so' movers'' [] ps'' agrfs'') -- IM
      _ -> ((ns,ps), agrfs, newmovers so' movers movers' ps' agrfs')                                -- EM
```

3 Agreement

```haskell
module Agr where
import MgTypes (F,Tier,Agr,Leaf,SO(L,S,E),typeVal,subList)
import Sel (hd)

agr :: SO -> SO
agr E = E
agr (L leaf) = L leaf
agr (S hso cso) = let (hso',cso') = (agr hso, agr cso) in
                  let (r,s,a) = hd hso' in
                  let (cso'',a') = iAgr cso' a in
                  let hso'' = putAgrInHead a' hso' in
                    S hso'' cso''
  where
  iAgr E a = (E, a)                              -- instantiate agr probe features, returning (newComp, newAgr)
  iAgr so ([],gs) = (so, ([],gs))
  iAgr so ((tier,f):ps,gs) = case nextLeafInTier tier so of
      Nothing -> let (so',(ps',gs')) = iAgr so (ps,gs) in (so', ((tier,f):ps', gs'))
      Just leaf -> let (f',so',changed) = iL so tier leaf f in
                   let (so'',(ps',gs')) = iAgr so' (ps,gs) in
                     if changed then (so'', (ps', f':gs')) else (so'', ((tier,f):ps', gs'))

  iL so tier (r,s,(ps,gs)) f =                   -- given probe f and leaf, search for value in leaf gs features
    let (ftype,fval) = typeVal f in let (fval',gs',changed) = matchAndFlag (ftype,fval) gs in
    if changed then (ftype++":"++fval', putAgrInTier tier (ps, gs') so, True) else (f, so, False)
    where
    matchAndFlag (_,fval) [] = (fval, [], False)
    matchAndFlag (ftype,fval) (g:gs) = let (gtype,gval) = typeVal g in
      if gtype == ftype
      then (gval, (gtype ++ ":_" ++ gval):gs, True)
      else let (gval',gs',changed) = matchAndFlag (ftype,fval) gs in (gval', g:gs', changed)

  nextLeafInTier _ E = Nothing
  nextLeafInTier tier (L h) = if leafIsInTier h tier then Just h else Nothing
  nextLeafInTier tier (S hso cso) = let h = hd hso in
    if leafIsInTier h tier then Just h else nextLeafInTier' tier (S hso cso)
    where -- after checking head of (S hso cso) above, now check heads-of-specs, then first-merged comp (if any)
    nextLeafInTier' tier (S (L _) cso) = nextLeafInTier tier cso
    nextLeafInTier' tier (S hso cso) = let ch = hd cso in if leafIsInTier ch tier then Just ch else nextLeafInTier' tier hso

  putAgrInTier tier a (L (r,s,a')) = L (r,s,a)
  putAgrInTier tier a (S hso cso) = let (r,s,a') = hd hso in
    if leafIsInTier (r,s,a') tier then S (putAgrInHead a hso) cso else putAgrInTier' tier a (S hso cso)
    where -- after checking head of (S hso cso) above, now check heads-of-specs, then first-merged comp (if any)
    putAgrInTier' tier a (S (L _) cso) = putAgrInTier tier a cso
    putAgrInTier' tier a (S hso cso) = let (r,s,a') = hd cso in
      if leafIsInTier (r,s,a') tier then S hso (putAgrInHead a cso) else S (putAgrInTier' tier a hso) cso

  putAgrInHead a (L (r,s,_)) = L (r,s,a)
  putAgrInHead a (S so so') = S (putAgrInHead a so) so'

  leafIsInTier leaf = foldr (\x sofar -> subList x (leafFeatures leaf) || sofar) False

leafFeatures :: Leaf -> [F] -- first positive sel feature ('category') and instantiated agr features (without goal-flagging)
leafFeatures (_,(_,p:_),(_,gs)) = p:map rmFlag gs where
  rmFlag f = let (ftype,fval) = typeVal f in if (not.null) fval && head fval == '_' then ftype++":"++tail fval else f
```

4 Head movement

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

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)) = let r' = hF (r,sel,agr) in
    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) = let r' = hF (r,sel,agr) in
    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')

  hF (r:rs,sel,agr) = joinStr "." (r:(head.snd) sel:snd agr):rs -- hm 'left adjunction' specifies features of moved heads
  hF ([],sel,agr) = []

rmHmDiacritics :: Morph -> Morph  -- for other modules
rmHmDiacritics = map (\x -> let y = case x of {'-':x' -> x'; _ -> x} in if (not.null) y && last y == '@' then init y else y)
```

5 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 (not.null) morph && head morph == "(" then L (morph,sel,agr) else L (["("]++morph++[")"],sel,agr)
```

6 Vocabulary insertion

```
module Vi where
import MgTypes (F,Morph,SO(L,S,E),subList,joinStr)
import Agr (leafFeatures)
import Hm (rmHmDiacritics)

vi :: SO -> SO
vi E = E
vi (S (L (["√de"],s,a)) (S (L (["√el"],s',a')) so)) = S (L (["del"],s,a)) (S (L ([],s',a')) (vi so))
vi (S (L (["√pres.T"],s,a)) (S (L (["√v*.Aux"],s',a')) so)) = S (L (["a"],s,a)) (S (L ([],s',a')) (vi so)) -- Fig 6b
vi (S so so') = S (vi so) (vi so')
vi (L (rs,s,a)) = L (m (rmHmDiacritics rs) (leafFeatures (rs,s,a)), s, a)

m :: Morph -> [F] -> Morph
m rs lfs = case rs of
  ("√past.T":_) -> mDefault ("BE":rs) lfs  -- for ex 53, Figure 5, English BE inserted
  _ -> mDefault rs lfs

mDefault :: Morph -> [F] -> Morph
mDefault rs lfs = case rs of
  ("√pres.T.φ:3s":rs') -> "-s":mDefault rs' lfs
  ("BE":"√past.T":rs') -> "was":mDefault rs' lfs -- for ex 53
  ("√past.T":rs') -> "was":mDefault rs' lfs
  ("√aux.Aux": "√pres.T" : rs') -> "sont":mDefault rs' lfs -- Fig 6a
  ("√elle":rs') -> if "φ:3p" `elem` lfs then "elle":"-s":mDefault rs' lfs else "elle":mDefault rs' lfs -- Fig 6a
  (r:rs') -> if subList ["N","φ:3p"] lfs then rDefault r:"-s":mDefault rs' lfs else rDefault r:mDefault rs' lfs
  [] -> []
  where
  rDefault cs = rmFs ("",case cs of {'√':cs' -> cs' ; _ -> cs})
  rmFs (t,f) = case f of {'.':_ -> t; c:cs -> rmFs (t++[c],cs); _ -> t}

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 = ""
```

7 Modular minimalist grammar, composed

```
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
```

Example session with ghci

As in the paper, lin puts moved constituents in parentheses, rather than deleting them, just
to make it easier to understand the analysis. To delete them, adjust the definition of del in Lin.hs.
Also note that smc is enforced here. To stop enforcing it, adjust definition of newmovers in Sel.hs.
Figure 1 = ex 8, Figure 2 is similar to the simpler ex 17-22, Figure 3 = ex 50, Figure 4 = ex 8,
Figure 5 = ex 53, Figure 6a = ex 58, Figure 6b = ex 66.

```
$ 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>
```