

Parsing with HPSG

6.863J Final Project

Alex Gruenstein

M.I.T. Computer Science and Artificial Intelligence Laboratory

May 16, 2005

1 Introduction

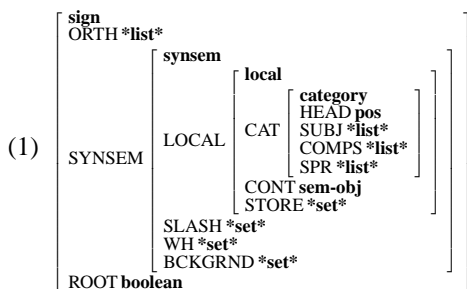
The goal of this project was to investigate the issues that arise when computationally implementing a well-developed syntactic/semantic theory. I have implemented a subset of the Head-driven Phrase Structure Grammar (HPSG) articulated in [2] [henceforth *G&S*] using the Linguistic Knowledge Building (LKB) system [1].

In this paper, I take the following path. First, I discuss briefly the theory as developed in *G&S*. Next I briefly describe the LKB and its capabilities. Finally, I describe my implementation of the grammar using the LKB, focusing particularly on areas which were challenging to implement.

2 Theoretical Background

G&S develop a grammar which addresses a lot of phenomena, focusing particularly on *interrogatives* and related long-distance dependencies. While there is no way I can do justice to their formalism in such a small space, I highlight here some of the salient features.

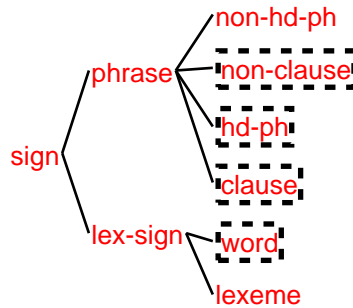
At a structural level, the grammar is defined in terms of *typed feature structures*; constraints on phrases are described in terms of *unification* with features of their daughters. Each word or phrase is represented as a *sign*, where a sign has the features shown in (1):



Items of particular interest are the following:

1. A *sign* is associated with the orthography ORTH of a word or phrase it describes.¹
2. A *sign* has both syntactic and semantic features associated with it under the SYNSEM feature.²
3. The CAT feature stores syntactic categorical information, including the part of speech (for phrases, typically inherited from the head daughter) and selection information in the form of SPR (specifier), SUBJ (subject), and COMPS (complements).
4. The CONT feature stores the semantic content of the sign. We will see this can offer quite interesting constraints later.
5. Most of the other features are used for long distance dependencies involving gaps and quantification: STORE, SLASH, and WH.

The *sign*, then, is the fundamental unit of interest. All other phrase and word types are descendants of *sign* in a multiple-inheritance hierarchy. At each level, constraints on the type are inherited from parent types – in addition, *default* inheritance is used in the type hierarchy, in which constraints can be inherited by default but *overridden*. Under *sign*, the tree diverges into *phrase* types and *lex-sign* types as shown below (dotted boxes simply indicate that I have hidden descendants of these types which I've implemented).



Most of the action in *G&S* is in the tree below *phrase*. The leaves of this tree are phrases which are used in the analysis (parse). *G&S* is concerned with defining this hierarchy so that the leaves come out just so, to give the parses they are after (and block parses of starred sentences). In section 4, I will flesh out many of these constraints in greater depth. Here, I simply point out that below *phrase* the tree has four nodes. Every phrase type later has as an ancestor two of these nodes: one of either *clause* or *non-clause* and one of either *hd-ph* (headed-phrase) or *non-hd-ph* (non-headed-phrase).

¹In *G&S* the feature PHON is used; I use ORTH instead as I deal with orthography rather than phonology in this project.

²In *G&S* signs also are associated with a CONTEXT feature to capture discourse contextual information. I have omitted this feature in my implementation and discussion. It is used in the resolution of fragments; a treatment which I do not address here.

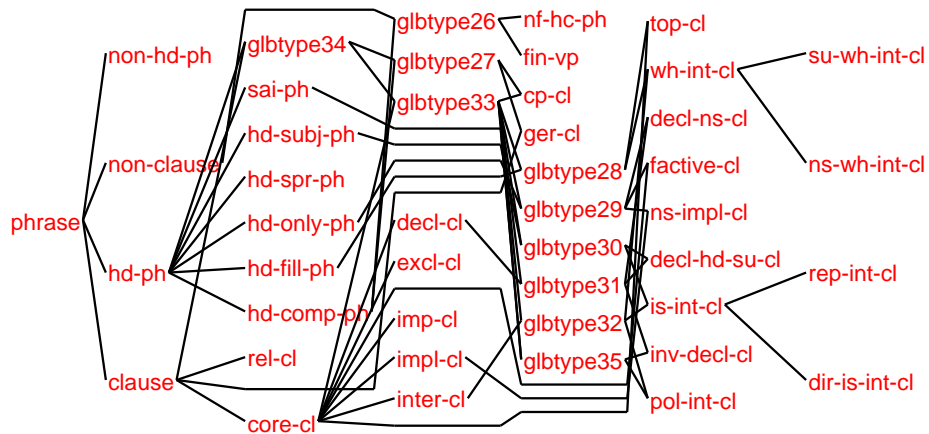
Hence, each phrase type is constrained along two dimensions: *headedness* and *clausality*.

Most phrases are a descendant of *hd-ph*, which describes two constraints: (1) The phrase has a *head daughter*, and (2) By default, the *SYNSEM* feature of the phrase is identified with (*i.e.* the same as) that of the head daughter (G&S call this the Generalized Head Feature Principle - GHFP). This constraint can be described as follows:

$$(2) \left[\begin{array}{l} \text{hd-ph} \\ \text{SYNSEM} / \square \\ \text{HD-DTR} [\text{SYNSEM} / \square] \end{array} \right]$$

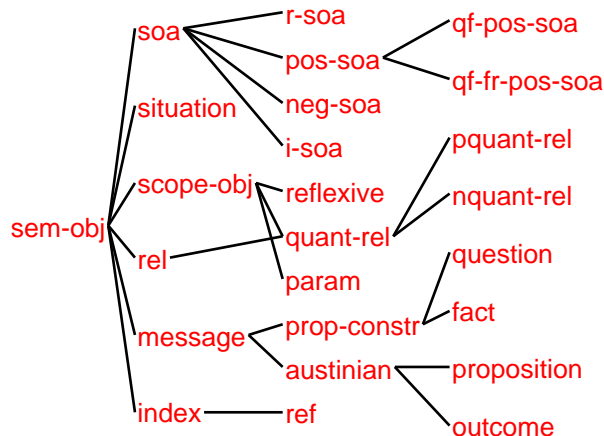
We note that \square indicates an identification of the two features, while the slash indicates that this is a default which can be overridden in subtypes.

I've given a flavor here of how the phrase type hierarchy works. The portion I've implemented (which is most of what is given in G&S) is shown below. It's a bit difficult to read, as the LKB will create artificial types called *glbtype* which are "greatest lower bounds" types. These are automatically inserted in the hierarchy where necessary for properly doing typed unification. The leaves of the hierarchy comprise the set of valid phrase types.



2.1 Semantics

I'll say a few quick words on semantics here. Semantics are, for the most part, represented in the *CONT* feature which is of type *sem-obj*. The following hierarchy describes the types of semantic objects available:



Of particular interest is the type *soa*, which represents a “state of affairs.” At its simplest, you can think of a *soa* as a relation ranging over objects. These objects are often indices named by NPs, or can also be *propositions* or *outcomes*, for example. A *proposition* or *outcome* has a *soa* embedded in it. So, in a declarative sentence, a *soa* is typically built up in the VP and then embedded in a *proposition* to form the CONT of an S.

3 LKB

The LKB is an environment for building grammars based on typed feature structures.³ To learn how to use it, I read [1] which describes the process of implementing grammars and provides several sample grammars. A grammar in the LKB consists of the following components:

1. A type hierarchy
2. Lexical entries
3. Grammar rules for building new phrases
4. A distinguished *start* type which specifies the constraints on a valid top-level phrase

The notation for defining these items is extremely straightforward: it is natural ASCII representation of typed feature structures. For example, to define the Generalized Head Feature Principle mentioned in the previous section, we simply write:

```

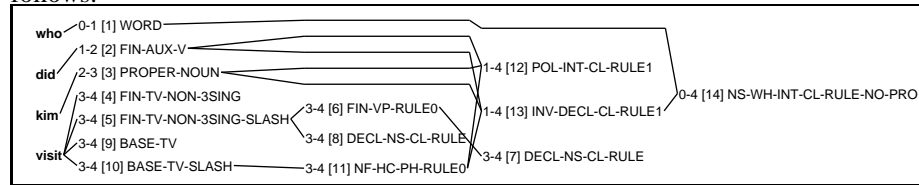
; GHFP
hd-ph := phrase &
      [ SYNSEM synsem /1 #1,
        HD-DTR [ SYNSEM synsem /1 #1 ] ].
  
```

³The LKB can be obtained from <http://lingo.stanford.edu>

This entry indicates that *hd-ph* inherits the constraints of *phrase* as well as adding the GHFP as a constraint. Lexical entries and rules are described with the same notation, though they have a slightly different meaning. While type definitions inherit constraints from super types, in lexical entries and rules you are creating *instances* of particular types. For instance, the following lexical entry creates an instance of a finite intransitive verb (*fin-iv*) with the ORTH feature instantiated:

```
smiled := fin-iv &
        [ ORTH <"smiled"> ].
```

The LKB also provides an interactive environment for investigating the type hierarchy, unifying types, viewing the parse chart, and so on. An example of a parse chart follows:



4 General Implementation Issues

In this section I describe my implementation, its coverage, and issues in turning theory into implementation.

4.1 Sets and Lists

The most pervasive difficulty in moving from theory to implementation was dealing with sets and lists. Constraints are frequently formulated in *G&S* which rely on choosing underspecified items from sets and lists. These can be quite complex. Consider, for example the *Store Amalgamation Constraint*:

$$(3) \left[\begin{array}{l} \text{word} \\ \text{SYNSEM.LOCAL} \left[\begin{array}{l} \text{CONT.QUANTS } \textit{order}(\Sigma_0) \\ \text{STORE}(\Sigma_1 \cup \dots \cup \Sigma_n) - \Sigma_0 \end{array} \right] \\ \text{ARG-ST} < [\text{STORE } \Sigma_1], \dots, [\text{STORE } \Sigma_n] > \end{array} \right]$$

This constraint actually licenses quite a few structures, requiring only that the QUANTS feature have a set of some objects which are not in the set of the STORE feature, which in turn is the union of the STORE features of the (potentially slashed) daughters of this phrase. Constraints such as these are quite powerful, however they can't be implemented directly in the LKB – as it does not support sets or set operations such as union.

Where possible, I implemented these constraints as best I could. Wherever I could get away with it, I used simple lists to replace sets. Lists are represented in the LKB as follows:

```
*list* := *top*.
*ne-list* := *list* &
```

```
[ FIRST *top*,
  REST *list* ].
```

Hence, a non-empty list (**ne-list**) is in the usual unification style first/rest notation.

Lists will only get you so far. It's not possible to concatenate two arbitrarily sized lists together. When this becomes necessary to simulate sets, we must turn to difference lists, defined as so:

```
*dlist* := *top* &
[ LIST *list*,
  LAST *list* ].
```

When using difference lists, LIST is a non-terminated list and LAST shares an index with the REST element of the first list. Thus, concatenating two lists can be accomplished as in the following rule:

```
inter-cl := core-cl &
          [SYNSEM.LOCAL [CONT question & [PARAMS [LIST #middle,
                                                  LAST #last]],
                        STORE [LIST #first,
                               LAST #middle]],
          HD-DTR [SYNSEM.LOCAL.STORE [LIST #first,
                                       LAST #last]]].
```

In general, I took two approaches to dealing with complex constraints involving sets and lists. When I knew I could get away with it and still deal with a large variety of phenomena, I reduced sets to lists – often assuming those lists would only have zero or one items. For instance, at the moment most of my SLASH rules for dealing with gaps allow only zero or one item to be slashed. For a first shot at a grammar, this covers a lot of phenomena.

The second approach came mostly in the lexicon. Many of the most interesting constraints are over words, as the constraint shown above. These constraints are meant to be productive, in the sense that they are supposed to give rise to several distinct lexical items. My implementation of the lexicon simply manually produces many of the different permutations allowed by the constraints, generating several lexical items. This is discussed below.

4.2 Parsers need rules!

A second major difference between theory and implementation is the fact that the LKB needs a set of *rules* to do parsing. While *G&S* define types suitable for representing phrases, the existence of these types are not sufficient for the LKB parser to figure out how to combine constituents together to make higher-level phrases. To accomplish this, it needs a set of rules.

My rules file is very simple. It basically consists of rules which inherit all of their constraints from the *leaves* of the phrasal type hierarchy. In some cases, these constraints are all that is necessary, as in the following definition of a rule for combining a word with its specifier:

```
hd-spr-rule := hd-spr-ph.
```

However, in some cases the constraints are not specific enough. For instance, in the *fin-vp* rule (for building finite verb phrases), the constraints on the type specify only that the verb have a subject followed by some number of complements. For the rule to work in the LKB, however, it must specify exactly how many complements will follow. Hence, to accommodate both intransitive and transitive verbs, I include the following two rules:

```
fin-vp-rule0 := fin-vp &
               [HD-DTR #h & [SYNSEM.LOCAL.CAT.COMPS <>],
               ARGS <#h>].
fin-vp-rule1 := fin-vp &
               [HD-DTR #h & [SYNSEM.LOCAL.CAT.COMPS <#1>],
               ARGS <#h, #1>].
```

For ditransitive verbs, a third rule would of course be needed. We note that *ARGS* specifies the daughters of this phrase (denoted by *DTRS* in *G&S*).

4.3 The *start* phrase

Here I describe the *start* or *top-level* phrase requirements (these basically define a valid *S*). Phrases must be able to unify with this type in order to be considered valid top-level parses. It is given as follows (from the file *start.tdl*⁴):

```
start := phrase & [SYNSEM [ LOCAL [CAT [HEAD verbal & [IC +,
                                                    VFORM fin],
                                                    SUBJ <>,
                                                    SPR <>,
                                                    COMPS <>],
                                                    CONT message,
                                                    STORE *null-dlist*],
                    SLASH <>,
                    WH *null-dlist*]].
```

The requirement is fairly straight forward: it must be a *verbal*, finite, independent (*IC +*) clause. It must not be expecting any subjects, specifiers, or complements and it must not have any slashed phrases. In addition, its *CONT* feature must be of type *message*.

4.4 Treatment of complements

Several other interesting compromises were made. Perhaps the most interesting, though subtle, involves the type used for *SUBJ*, *SPR*, and *COMPS*. In *G&S* these are lists of type *synsem*. As these come from the daughters of the phrase, *G&S* rely on a function *signs-to-synsems* to extract them from the daughters, as in the following:

$$(4) \left[\begin{array}{l} \mathbf{hd-comp-ph} \\ \text{HD-DTR } \square \left[\begin{array}{l} \mathbf{word} \\ \text{COMPS } \textit{signs-to-synsems}(\textit{nelist}(\square \oplus \textit{list})) \end{array} \right] \\ \text{DTRS } < \square \oplus \square > \end{array} \right]$$

⁴I've incorporated the empty comps constraint (ECC) that applies to all phrases here, though technically it inherits it from *phrase*.

There is, of course, no way to do unification in the parser while incorporating the *signs-to-synsems* function. One option would be to move any constraints dealing with non-singleton lists of daughters to the rules file, and writing several rules in each case to deal with multiple possibilities. This would lead to a larger number of rules, and perhaps wouldn't be so bad. The other approach, which I've taken here, is just to unify in the entire daughter, not just its SYNSEM feature. This results in parses which are not as elegant, but which get the job done. So, for example, my *hd-comp-ph* constraints are defined as follows, to directly unify in a portion of the daughters list to the COMPS:

```
hd-comp-ph := hd-ph &
             [HD-DTR #1 &
              word &
              [SYNSEM.LOCAL.CAT.COMPS #A],
              ARGS [FIRST #1,
                   REST #A]].
```

5 Parsing Coverage

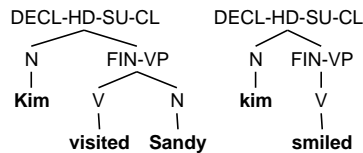
I've explained some of the technical details in moving from theory to implementation. Now I turn to discussing some of the sentences I can parse.

5.1 Declarative Sentences

To get things rolling, let's first turn to some simple declarative sentences:

- (5) a. Kim smiled
- b. Kim visited Sandy
- c. Kim visits Sandy
- d. *Kim visit Sandy
- e. *Kim smiled Sandy

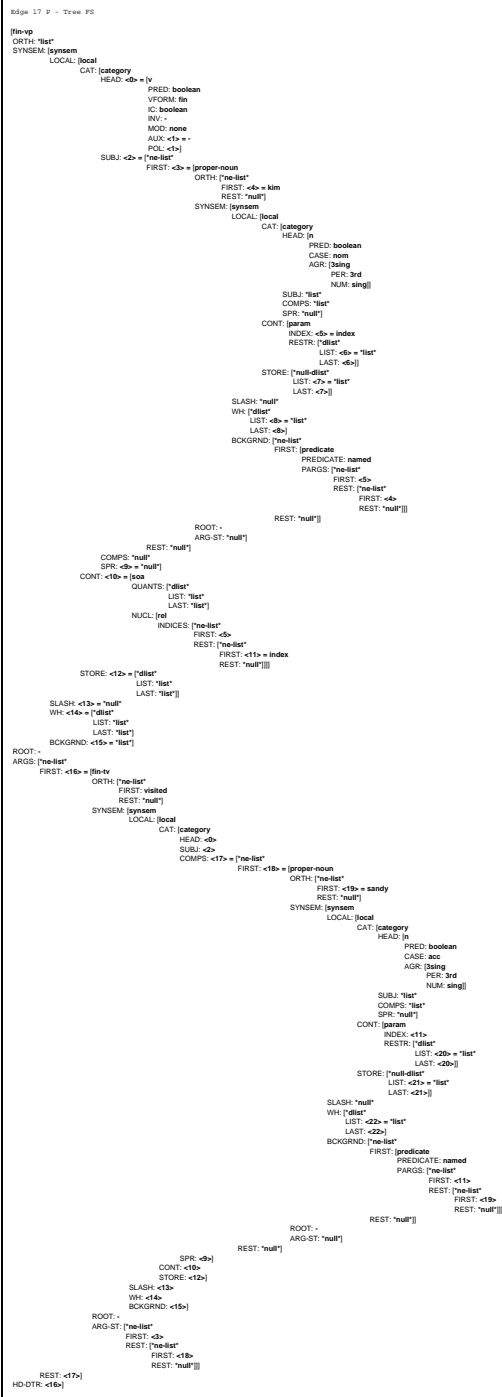
The parse trees for (5a) and(5b) are as follows:



We note that these are brief representations, each node is in fact quite a large feature structure: For instance the node labeled *fin-vp* representing *visited sandy* corresponds to the feature structure shown in figure 1.

First, let's look at the definition of the *fin-vp* phrasal type which is used to represent *visit sandy*:

Figure 1: The *fin-vp* for visited sandy



```

fin-vp := non-clause & hd-comp-ph &
  [SYNSEM.SLASH #S,
   HD-DTR #h & [SYNSEM [LOCAL.CAT [ HEAD [VFORM fin,
                                     AUX #1,
                                     POL #1],
                                     COMPS #C],
                 SLASH #S]],

   ARGS [FIRST #h,
         REST #C]].

```

We note it inherits from *non-clause* and *hd-comp-ph*. The former doesn't add any constraints, the latter inherits the GHFP from *hd-ph* discussed above and does not stipulate any other constraints not made more specific in this type. Informally, *fin-vp* is just what you would expect: it is a phrase headed by a finite verb which takes zero or more complements. Lexically, verbs specify the sorts of complements they take, as well as the proper semantics. So, for instance, a fully fleshed out lexical entry for a verb like `visits` would look like the following:

```

visits := word &
  [ORTH <'visits'>
   SYNSEM.LOCAL [CAT [HEAD v & [VFORM fin],
                        SPR <>,
                        SUBJ <#1>,
                        COMPS <#2>],
                 CONT soa & [NUCL rel & [INDICES <#i,#j>]]],
   ARG-ST <#1 & [SYNSEM.LOCAL [CAT.HEAD n & [CASE nom,
                                               AGR [PER 3rd,
                                               NUM sg]],
                               CONT [INDEX #i]],
                #2 & [SYNSEM [LOCAL [CAT.HEAD n & [CASE acc],
                                               CONT [INDEX #j]],
                               WH <! !>]]>].

```

We note that the CONT feature will be projected to the *fin-vp*, as `visits` will be the head of the phrase. Thus, semantically the VP will be a *soa* (“state of affairs”) relation between the indices of the subject NP and the complement NP.⁵ Finally, we note that the subject must have proper agreement

Next, let's look at *decl-hd-su-cl* which puts together the subject, Kim and the *fin-vp* visited Sandy. This type does pretty much what you would expect syntactically. What's interesting about it is the constraints it inherits from *decl-cl*:

```

decl-cl := core-cl &
  [SYNSEM.LOCAL.CONT austinian & [SOA soa /L #1],
   HD-DTR [SYNSEM.LOCAL.CONT soa /L #1]].

```

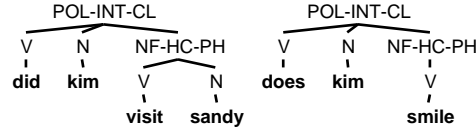
This constraint says that by default (hence the /L), we take the *soa* from the head daughter and embed it in an *austinian*. This is compatible with the semantic requirements for the *start* phrase. In contrast, the *fin-vp* in this case will be blocked from being a *start* phrase because (1) it has not discharged its subject, and (2) its CONT is of type *soa*.

⁵I've simplified the semantics a bit in my implementation. While my semantics require only a relationship between the indices, *G&S* require that the relation be of type *visit-rel*. This is not necessary to get proper parsing, so I've punted..

5.2 Polar Questions

Let's turn now to polar questions:

- (6) a. Did Kim visit Sandy?
 b. Does Kim smile?



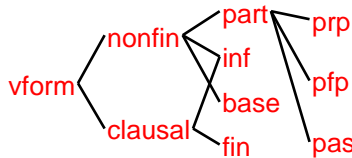
We note that at the top level of the parse we have *pol-int-cl* (polar interrogative clause). In addition, since our VP is not finite, we do not use *fin-vp*; instead, we use the type *nf-hc-ph*. It is the same as *fin-vp* except that it requires a nonfinite head verb:

```
nf-hc-ph := non-clause & hd-comp-ph &
          [SYNSEM.SLASH #S,
           HD-DTR #h,
           ARGS [FIRST #h & [SYNSEM [LOCAL.CAT [HEAD.VFORM nonfin,
                                                COMPS #C],
                                                SLASH #S]],
                REST #C]].
```

To satisfy this, we will need a lexical entry like the following:

```
visit-base := word &
            [ORTH <'visit'>
             SYNSEM[LOCAL [CAT [HEAD v & [VFORM base],
                                     SPR <>,
                                     SUBJ <#1>,
                                     COMPS <#2>],
                           CONT soa & [NUCL rel & [INDICES <#i,#j>]]]]
             ARG-ST <#1 & [SYNSEM.LOCAL [CAT.HEAD n & [CASE nom],
                                         CONT [INDEX #i]],
                           #2 & [SYNSEM [LOCAL [CAT.HEAD n & [CASE acc],
                                         CONT [INDEX #j]],
                                         WH <! !>]]>].
```

We note that in this case the VFORM of the verb is *base*. That is, the baseform of the verb (baseforms can be selected by *to* to form a phrase of type *inf*. This might be a good place to show the *vform* hierarchy:



What's more interesting is how the *pol-int-cl* is built at the top level. The definition of *pol-int-cl* is shown below, as well as for *sai-ph* and *inter-cl* which it inherits directly from and also provide relevant and interesting constraints here. We note that *sai-ph* forces the head daughter to be INV + and AUX + thus requiring an auxiliary verb and indicating an inverted phrase. The constraints on *inter-cl* (interrogative-clause) are mainly interesting in that they make the CONT a *question*. Finally, *pol-int-cl* requires that that the question be about a *proposition* constructed from its complement VP.

```

pol-int-cl := inter-cl & sai-ph &
  [SYNSEM.LOCAL.CONT [PARAMS *null-dlist*,
    PROP proposition & [SOA #1]],
  HD-DTR #h & [SYNSEM.LOCAL [CAT.HEAD.IC +,
    CONT #1]],
  ARGS [FIRST #h,
    REST *list*]].

sai-ph := hd-ph &
  [SYNSEM.LOCAL.CAT.SUBJ <>,
  HD-DTR #h & word & [SYNSEM.LOCAL.CAT [HEAD [INV +,
    AUX +],
    SUBJ <#0>,
    COMPS #A]],
  ARGS [FIRST #h,
    REST [FIRST #0,
    REST #A]]].

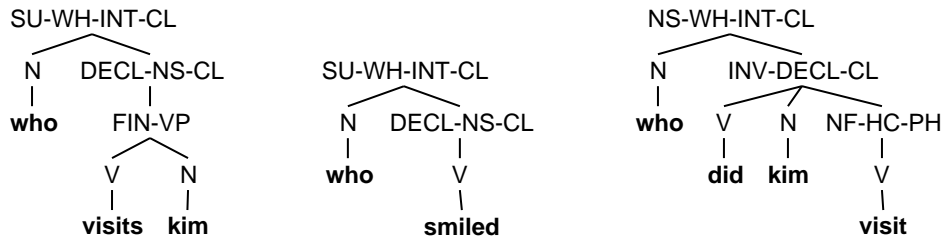
inter-cl := core-cl &
  [SYNSEM.LOCAL [CONT question & [PARAMS [LIST #middle,
    LAST #last]],
  STORE [LIST #first,
    LAST #middle]],
  HD-DTR [SYNSEM.LOCAL.STORE [LIST #first,
    LAST #last]]].

```

5.3 *wh*-Interrogatives

We now turn to *wh*-interrogatives. These questions are interesting because they introduce long-distance dependencies.

- (7) a. Who visits Kim?
- b. Who smiled?
- c. Who did Kim visit?
- d. *Who did Kim visit kim?
- e. *Who did smile?



To deal with the subject gaps, two new types of phrases are introduced: *su-wh-int-cl* and *decl-ns-cl*. The latter is mainly interesting in that it inherits from *decl-cl*, which as

discussed above takes a *soa* and embeds it in a *austinian* (a supertype of *proposition*, which is what we are after here). It adds the additional constraints that it must be a non-inverted VP. In this way, we get semantics appropriate for our top-level phrase – however, we don’t have a top level phrase yet as the SLASH feature will be non-empty since we have not found a subject yet.

```
decl-ns-cl := decl-cl & hd-only-ph &
            [SYNSEM.LOCAL.CAT [HEAD v & [INV -],
                               SUBJ < phrase >]].
```

The phrase *su-wh-int-cl* (subject wh interrogative clause) then takes this propositional content and embeds it in a question, while at the same time unifying the slashed NP in with a wh-NP. This is rather complicated and done through a web of inheritance from interrogative clauses and head filler phrases. The most interesting relevant types are given below:

```
su-wh-int-cl := wh-int-cl &
               [SYNSEM.LOCAL.CAT.SUBJ <>,
                HD-DTR #h,
                ARGS <[SYNSEM.LOCAL #4],
                     #h &
                     [SYNSEM.LOCAL.CAT.SUBJ <[SYNSEM gap-ss & [LOCAL #4]]>>]].
```

```
wh-int-cl := inter-cl & hd-fill-ph &
            [SYNSEM.LOCAL.CONT.PARAMS <! #1 !>,
             HD-DTR #h,
             ARGS <[SYNSEM.WH <! #1 !>], #h>] &

            [SYNSEM.LOCAL.CONT.PROP #2,
             HD-DTR [SYNSEM.LOCAL.CONT #2]].
```

```
hd-fill-ph := hd-ph &
            [SYNSEM.SLASH *null*,
             HD-DTR #h,
             ARGS < [SYNSEM.LOCAL #1],
                   #h & phrase & [SYNSEM [LOCAL.CAT.HEAD v,
                                           SLASH < #1 >]] >]].
```

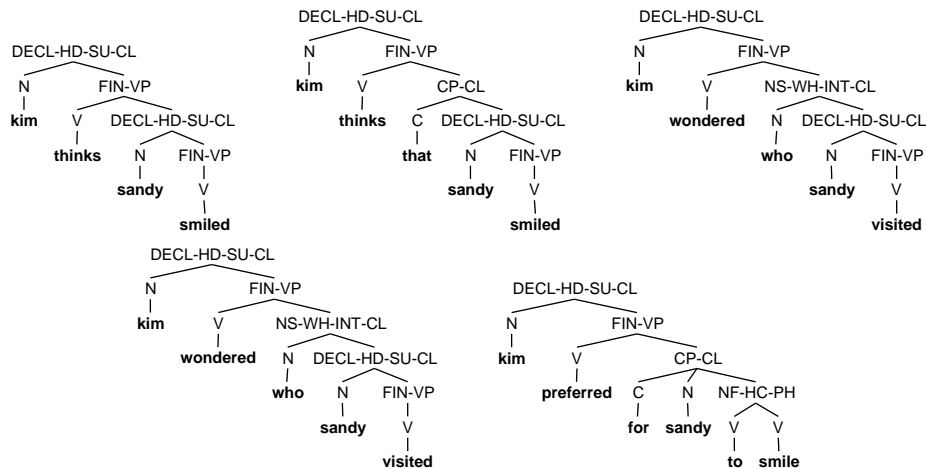
```
inter-cl := core-cl &
          [SYNSEM.LOCAL [CONT question & [PARAMS [LIST #middle,
                                                  LAST #last]],
                       STORE [LIST #first,
                              LAST #middle]],
          HD-DTR [SYNSEM.LOCAL.STORE [LIST #first,
                                      LAST #last]]].
```

For *non-subject* gaps, as in *who did kim visit* we introduce a similar web of new phrases. I won’t go into the details here, but a similar mechanism of passing along gaps is made. The complex type hierarchy is put to use, as many of the constraints are shared across both types of wh-questions.

5.4 Verbs with non-NP complements

I now turn to sentences involving verbs with non-NP complements. Here are some examples:

- (8) a. Kim thinks Sandy smiled.
 b. Kim thinks that Sandy smiled.
 c. *Kim thinks who visited Sandy. [unstressed *who*]
 d. Kim wondered who visited Sandy.
 e. Kim wondered who Sandy visited.
 f. *Kim wondered Sandy smiled
 g. Kim preferred for Sandy to smile



These examples are all handled in the lexicon, and almost entirely through the CONT feature. Each selects a verbal phrase with the appropriate type of *sem-obj*: *question*, *proposition*, or *outcome*. There's also a *cp-cl* for phrases involving *that* and *for...to*. I won't go into the details of all of those here, however the complementizers *that* and *for* are relatively interesting, as is the lexical entry for *to*, so I've shown them below. We note that *for* and *that* form complementizer phrases which are IC - indicating they are not independent clauses (and hence can't be top level). The entry for *to* allows the creation of infinitival phrases.

```
that := word &
[ORTH <"that">,
  SYNSEM [LOCAL [CAT [HEAD c & [IC -,
    VFORM fin],
    COMPS <#2>],
  CONT #1 & austinian]],
  ARG-ST <#2 & [SYNSEM.LOCAL [CAT [HEAD [IC +,
  VFORM fin],
```

```

SUBJ < >],
  CONT #1]]>].

for := word &
  [ORTH <"for">,
  SYNSEM [LOCAL [CAT [HEAD c & [IC -,
    VFORM inf],
    SUBJ <>,
    COMPS <#3, #4>],
    CONT outcome & [SOA #1]],
  WH <! !>],
  ARG-ST <#3 & [SYNSEM canon-ss & [LOCAL #2]],
  #4 & phrase & [SYNSEM.LOCAL [CAT [HEAD.VFORM inf,
    SUBJ <[SYNSEM.LOCAL #2] >],
    CONT #1]]>].

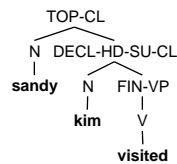
to := word &
  [ORTH <"to">,
  SYNSEM.LOCAL [CAT [HEAD v & [AUX +,
    VFORM inf],
  SUBJ <#7 & [SYNSEM [LOCAL #2,
    SLASH < >]]>],
  CONT #1],
  ARG-ST <#7, phrase & [SYNSEM.LOCAL [CAT [HEAD.VFORM base,
  SUBJ <[SYNSEM.LOCAL #2] >],
  CONT #1]]>].

```

5.5 Topicalization

Finally, I give an example involving SLASH: topicalization. Consider the following example in which the object of *visited* has been moved to the front of the sentence.

(9) Sandy Kim visited



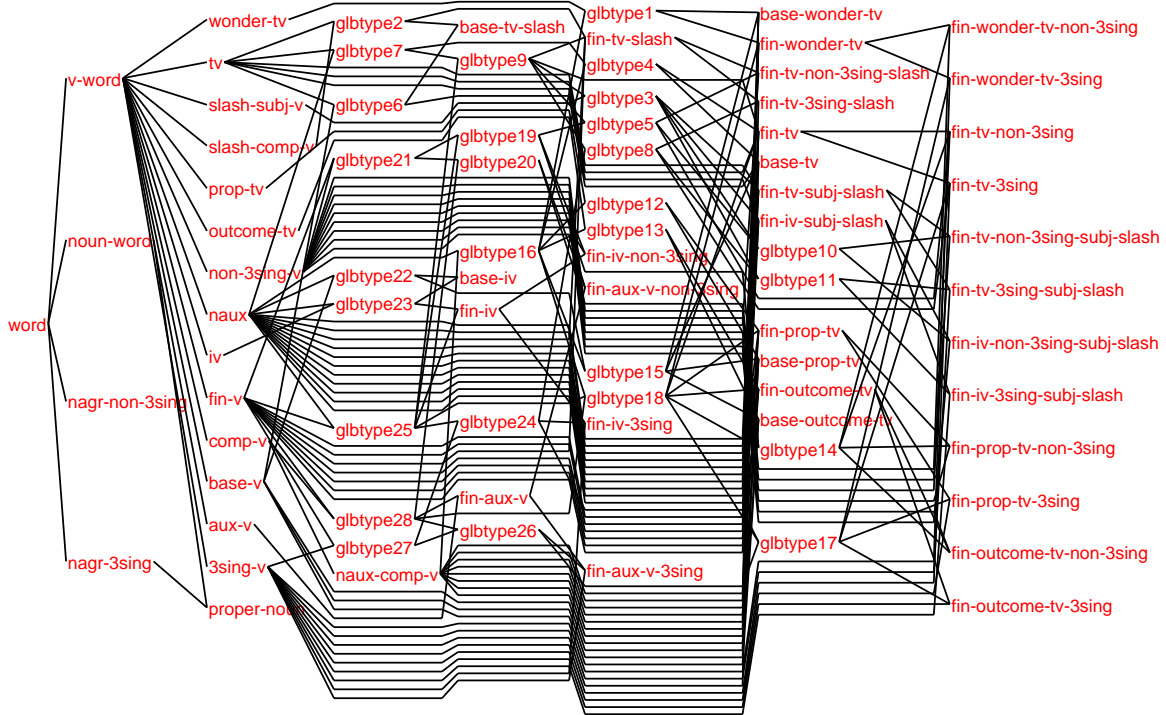
Here, *top-cl*, allows the slashed complement to be filled in with the topicalized NP *sandy*.⁶

6 Lexicon

I now turn briefly to describing the lexical approach I developed. In *G&S* the lexicon is only marginally dealt with - constraints over words are formulated, however as I

⁶I'm still working out some bugs with *top-cl*, right now it is a little too permissive

discussed above these constraints are meant really to be lexically *productive* rather than as true constraints. I have taken a fairly simple approach to the lexicon. The really proper way to do things would be to have lexemes and lexical+morphological rules over these lexemes to produce *words*. Given my time limitations, I instead use a multiple inheritance hierarchy to make the different word types I need. The hierarchy I developed is as follows:



Let's focus on verbs. I specify verbs along five dimensions:

1. number of arguments
2. slashed
3. finite/base
4. agreement features
5. auxiliary/not auxiliary

All verb types inherit from a *v-word* which requires at least a subject:

```
v-word := word &
        [SYNSEM.LOCAL [CAT [HEAD v,
                           SPR <>,
                           SUBJ <#1>],
                           CONT soa & [NUCL rel & [INDICES [FIRST #i]]]],
        ARG-ST <#1 & [SYNSEM.LOCAL [CAT.HEAD n & [CASE nom],
                                     CONT [INDEX #i]], ...>].
```

To demonstrate how I specify constraints along the five dimensions enumerated above, let's look at an example. We'll see all the types involved in making an entry for a non-auxiliary, finite, transitive, non-slashed verb, that has *3sing* agreement:

Transitivity:

```
tv := v-word &
    [SYNSEM.LOCAL.CONT.NUCL.INDICES <index, #j>,
     ARG-ST <*top*, [SYNSEM [LOCAL [CAT.HEAD n & [CASE acc],
     CONT [INDEX #j]],
     WH <! !>]]>].
```

A non auxiliary verb is one that is AUX -:

```
naux := v-word & [SYNSEM.LOCAL.CAT.HEAD.AUX -].
```

A verb that does not slash its complements:

```
comp-v := v-word &
    [SYNSEM [LOCAL.CAT [COMPS <#2>]],
     ARG-ST <*top*, #2>].
```

It's finite:

```
fin-v := v-word &
    [SYNSEM.LOCAL.CAT.HEAD.VFORM fin].
```

And, finally, it has agreement constraints:

```
3sing-v := v-word &
    [ARG-ST [FIRST [SYNSEM.LOCAL.CAT.HEAD.AGR 3sing]]].
```

We put all of these together as follows:

```
fin-tv := tv & fin-v & naux-comp-v & naux.
fin-tv-3sing := fin-tv & 3sing-v.
```

And then we can make a simple lexical entry:

```
visits := fin-tv-3sing &
    [ORTH < "visits" >].
```

7 Conclusion

Even though *G&S* formulate a well-developed and relatively bug-free theory, implementation was still fairly daunting. It is very difficult to get all of the constraints just-so: while the theory is very solid at a general level, there are many implementation details that must be gotten exactly right for everything to work out, as discussed above. The complex web of inheritance in *G&S* is fantastic for introducing new types of phrases, but it is challenging to implement. Compromises must sometimes be made during the implementations, and when these compromises are made at high levels in the inheritance hierarchy, their implications at the leaves can be hard to envision and correct for. While the hierarchy allows parsimony in writing rule writing, it makes debugging fairly difficult as you must search up the hierarchy to find bugs that result in erroneous parses.

References

- [1] Ann Copestake. *Implementing Typed Feature Structure Grammars*. CSLI Publications, Stanford, CA, 2002.
- [2] Jonathan Ginzburg and Ivan A. Sag. *Interrogative Investigations: the Form, Meaning and Use of English Interrogatives*. CSLI Publications, 2000.