Computational Linguistics II (Fall 2006, Exercise 2)

High-Level Goals

- Deploy co-references in agreement and phrase structure recursion.
- Implement an analysis of pre- and post-head modification.
- Eliminate redundancy from the lexicon by the use of types.

Background Reading

Read Sections 4.1 through 4.6 from Sag, Wasow, & Bender (2003). Make sure to compare their examples and grammars to our slide copies and observe where in the lecture we have in some cases (further) simplified over the book. Although Sag, Wasow, & Bender (2003) seem to advocate a *flat* VP structure—i.e. a lexical head combining with all of its complements through a single rule application—see whether you can find a discussion of the analysis we adopt (a binary branching, recursive VP) somewhere in the text. Furthermore, compare their tree in (26) to the tree derived by our grammar for a comparable sentence, e.g. *the dog chased the cat*. What are the differences, if any?

1 Obtaining the Starting Grammar

- Like last week, login to the Linux server '`mt.ifi.uio.no`' through X Windows.
- If you completed the previous exercise and are fully comfortable with your solution, you can continue from the contents of your '`grammar1`' directory. If, however, you would rather get a fresh start for this exercise, you can obtain a new starting grammar, equivalent to the model solution for the previous exercise, by typing at the shell prompt:

  ```
  cvs checkout grammar2
  ```

  Note that, in case you get a fresh grammar, it will be in a new directory called '`grammar2`'.
- Launch *emacs* from the shell ('`emacs &`'), and within emacs, start up the LKB ('`M-x lkb RET`'), then load the grammar using the LKB menu Load | Complete grammar.

2 Use of Feature Structure Re-Entrancies in Agreement (10 Points)

- Since the feature `AGR` is introduced on the type *pos*, all kinds of words will have an agreement feature. However, in English only determiners and nouns (and probably verbs, depending on which perspective one takes) have agreement information of their own. Unused features unnecessarily increase the size of the grammar and can make errors more difficult to track down. Modify your grammar so that the feature `AGR` only appears on `AGR`-bearing *pos* subtypes and not on others like *prep*. To do this, you will need to add a new type, say *agr-pos*, below *pos* and then make *det*, *noun*, and *verb* subtypes of the new *agr-pos*.
- The intuition behind determiner – noun agreement is that the `AGR` value of the noun must be the same as that of its specifier. In our lexicon, though, the `AGR` value of the noun and the `AGR` value of its specifier are stipulated separately. Use re-entrancies to eliminate this redundancy and capture the generalization underlying agreement, viz. that the `AGR` value of the noun itself is *identical* to that of its specifier. As always, verify your changes by parsing your test sentences, specifically the ones testing agreement.
- In case the syntax of re-entrancies in TDL is not clear yet, here's an example:

  ```
  x := y &
  [ F #1 & z,
    G #1 ].
  ```

  This definition means that the value of the feature `F` is z, and the value of the feature `G` is the same as that of the feature `F`.

3 Phrase Structure Recursion (20 Points)

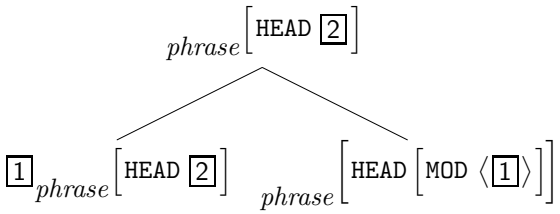
- As the LKB (and similar parsing systems) assume that each rule has a fixed number of daughters, we ended up with two instances of the head – complement schema: one binary-branching, picking up a single complement, the other ternary, picking up two complements (plus, strictly speaking, the third instance, which is unary and promotes intransitive lexical heads into phrases).
- To condense the binary and ternary instances of the head – complement schema into just one, investigate a binary-branching VP analysis. For a ditransitive head (e.g. *gave*), there will need to be two consecutive applications of the head – complement rule, each picking up one complement at a time and reducing the `COMPS` list appropriately. Use the batch parsing machinery to make sure that there is no spurious ambiguity.

4 An Analysis of 'Free' Modifiers (20 Points)

- So far, we have grammar rules to combine heads with two general types of sister constituents, viz. complements to the right of the head and specifiers to its left. Extend the grammar to provide an analysis of modification, admitting sentences like *the dog barks near the cat*.

  We introduce a new head feature `MOD` and a new syntactic rule for modifiers, and we make use of the notion of underspecification.

- In the file '`types.tdl`', add the feature `MOD` to the definition of the type *pos*, with its value constrained to be of type **list**, the same type as for the `SPR` attribute.
- Also in the types file, assign an appropriate value for `MOD` to each of the subtypes of *pos*. In our analysis of 'free' modification, modifiers use a feature structure of type *expression* inside of their `MOD` list to constrain the type of phrase they can attach to. In other words, modifiers are *not* selected for by head daugthers, but instead the modifier daughter is the one to select which heads it can modify. Non-modifiers, i.e. everything but prepositions at this point, will have an empty `MOD` list. Prepositions, on the other hand, should have exactly one element in their `MOD` list, effectively constraining which phrases they will be able to modify.
- Add *near* as an additional preposition in the lexicon, copying the entry of *to* and adapting it as needed.
- In the file '`rules.tdl`, add a new head – modifier rule *somewhat* similar to the existing specifier – head rule, but with the modifier daughter constraining the head daughter, e.g.



- In addition to the above constraints, determine the `SPR` and `COMPS` values on the mother. Save your changes, then test your revised grammar using the test file '`mod.items`'. Examine the results, and make any necessary corrections.
- If your analysis does not already admit examples like *the dog near the cat barks*, modify your grammar appropriately to also allow prepositions to modify nominal phrases. In order for modifiers to select for phrases headed either by a verb or a noun, consider the introduction of an additional type *modable* into the *pos* hierarchy, such that *verb* and *noun* will both be compatible with the new *modable*.
- If your analysis provides two parses for the sentence *the dog barks near the cat*, modify your grammar to eliminate one of the two parses, then run the batch parse again with the file '`mod.items`', and examine the results.
- Add additional sentences to the file '`mod.items`', and notice what happens to the number of analyses as you add several prepositional phrase modifiers within a single sentence.

5 Selecting Specific Prepositional Phrase Complements (10 Points)

- We still fail to enforce the requirement on the PP complement of (the second lexical entry for) *gave* that the PP be headed by the specific preposition *to*. Add an additional attribute `PFORM` to the type *prep* and constrain it to take a value of type **string**. Next, in '`lexicon.tdl`', make sure that the value of `PFORM` (inside of *prep*) appropriately reflects the surface form of the preposition, e.g. *"to"* for the lexical entry *to*.

- Adding `PFORM` to *prep* has the side-effect of making it a `HEAD` feature, i.e. one of the properties that will be automatically passed up from a head daughter to its mother, e.g. from a preposition to the PP built from combining a prepositional head with its NP complement. Parse the sentence *the cat gave the dog to the animal* (using your own animal, of course; or ours, *aardvark*, if you started with a fresh grammar today). Inspect the feature structure of the PP node in the parse tree and confirm that its `PFORM` value is indeed *"to"*.

- At this point, all PPs in our trees will be marked for `PFORM`, such that lexical heads selecting for PP complements can now require a specific value. Add the necessary constraint to *gave* in the lexicon.

6 The Head Feature Principle (15 Points)

- Looking at the various rules, you will have noticed that in each rule the `HEAD` value of the whole phrase is always the same as the `HEAD` value of one of the daughters in `ARGS`. The argument which contributes the `HEAD` value to the whole phrase is known as the *head daughter* of the phrase. For some kinds of phrases, the head daughter is the first daughter, and for some it's the last daughter. Rearrange the hierarchy of rules to capture this distinction between head-initial phrases and head-final phrases.

- In '`types.tdl`', add three new types:

```
head-initial := phrase &
[ HEAD #head,
  ARGS [ FIRST [ HEAD #head ] ] ].

head-final := phrase &
[ HEAD #head,
  ARGS < expression, [ HEAD #head ] > ].

root-head-final := root & head-final.
```

  Note that our definition of *head-final* makes the simplifying assumption that all head-final phrases are binary, i.e. have exactly two daughters (which is true for our current grammars). Also, we will have more to say about the type *root-head-final* later in the course.

- Modify the rules in '`rules.tdl`' to inherit from these new types. For example, the *head-complement-rule-0* should look like:

```
head-complement-rule-0 := head-initial &
[ SPR #spr,
  COMPS < >,
  ARGS < word &
         [ SPR #spr,
           COMPS < > ] >  ].
```

  We call this rule head-initial, even though it has only one daughter, since the other head – complement rules are also head-initial. Head-final rules, like the *head-specifier-rule*, should inherit from the type *head-final*. The feature `HEAD` should not need mentioning in '`rules.tdl`' at all, except for one occurrence in the head – modifier rule perhaps.

7 Eliminating Redundancy in the Lexicon (25 Points)

- Using the same strategy, i.e. the introduction of additional types for common feature structure configurations, find and eliminate more redundant specifications in the grammar. Improve the organization of the type hierarchy to make it easier to add new words that are similar to words already in the lexicon. As a place to start, take a look at the lexical entries for nouns, and note that the same information is stated again and again in each entry. Recast those generalizations as constraints on a new type, *noun-word*, which every noun lexical entry inherits from. As you work, use the batch parse facility now and again to make sure none of your modifications have damaged the coverage of the grammar.
- Further eliminating redundancy from the lexicon, introduce subtypes of the type *word* for determiners, verbs, and other parts of speech, adding any constraints that are true for all instances of each class, e.g.

  ```
  det-word := word & [ ... ].
  ```

- Introduce subtypes of the *noun-word* type for singular and plural nouns, and do the same for determiners.
- Introduce subtypes of the *verb-word* type whose instances select for third-singular or non-third-singular subjects for present-tense verbs, and an additional subtype of the verb-word type for past-tense verbs.
- Introduce subtypes of the *verb-word* type to distinguish intransitives, transitives, and the two types of ditransitive verbs.
- Take advantage of the notion of multiple inheritance to introduce lexical types in the file '`types.tdl`' for the verbs in the file '`lexicon.tdl`', making use of types from each of these two sets of subtypes of the verb-word type. Remember that the syntax for defining multiple inheritance in TDL is as follows:

  ```
  x := y & z & [A b].
  ```

  This definition says that the type x is a subtype of both y and z, and x introduces the feature `A` with value b. Note that lexical entries in the file '`lexicon.tdl`' can only inherit from a single type, so any multiple inheritance that you introduce must be defined in the types file.
- Modify your entries in the file '`lexicon.tdl`' to make use of these new types. When you are finished with this exercise, each of the definitions in your '`lexicon.tdl`' file should consist of the name of the lexical entry, the name of its lexical type, and the orthography. Everything else should be defined in the file '`types.tdl`' file. Here is a sample ideal entry:

  ```
  dog := noun-word-3sing &
  [ ORTH "dog" ].
  ```

8 More Modification (Optional; 20 Points))

- Add an analysis for attributive adjectives like *angry* and *fierce*, as in the example *those fierce dogs bark*. Introduce a new subtype of *pos* called *adj*. As with each extension of the grammar, design your analysis to admit only grammatical sentences, avoiding both overgeneration (blocking, for example, **those dogs fierce bark*) and undergeneration (admitting, for example, *those fierce angry dogs bark*). You will likely need a type distinction between pre- and post-modifiers, i.e. adjectives vs. prepositional phrases. Aim to incorporate this additional distinction into the existing *pos* hierarchy, rather than adding a new feature.
- If possible, try to avoid spurious ambiguity (i.e. multiple 'equivalent' analyses for the same input) as, for example, in *Those fierce dogs near the cat bark*. Test your analysis using the batch parse facility on the file '`adj.items`', and make adjustments as needed. Add some additional items to this file to illustrate the effects of your analysis.
- Extend the grammar to admit adverbs like *happily*, which modify verbal projections (verb phrases and sentences). Introduce a new subtype of *head* called *adv*. Notice that adverbs can appear both before and after the phrase they modify: *the dog barks happily* but also *the dog happily chased the cat*. Test your analysis using the file '`adv.items`', and tune your changes as needed. Then re-run the batch parse using the files '`mod.items`' and '`adj.items`' to make sure that your previous analyses are still healthy.

Submit your files in email to Dan and Stephan no later than 09:00 on Tuesday, September 26.