Computational Linguistics (Spring 2011) — Exercise 2b

High-Level Goals

- Become familiar with the LKB grammar development system.
- Learn to extend the grammar by adding lexical entries.
- Implement a unification-based account of agreement constraints.
- Implement an analysis of pre- and post-head modification.
- Eliminate redundancy from the lexicon by the use of types.

1 Starting the LKB — Analyzing Sentences (0 Points)

For our practical exercises, we will be using a specialized software tool called the Linguistic Knowledge Builder (LKB); this is installed on all Linux machines at IFI. Many of the IFI students laboratories are installed with Linux, (including our laboratory room on Thursdays this semester. We will make available instructions for how to connect to the IFI Linus environment when working from home or using a workstation (at IFI or elsewhere) running Windows.

(a) To prepare your account for use in this class, you need to perform a one-time configuration step. At the shell command prompt (on an IFI Linux machine), execute the following command:

```
~oe/bin/inf2820
```

This command will add a few lines to your personal start-up file '`.bashrc`'. For these settings to take effect, you need to log out one more time and then back in. Once complete, you need not worry about this step for future sessions; the additions to your configuration files are permanent (until you revert them one day, maybe towards the end of the semester).

(b) Launch the LKB grammar development that we will use throughout the semester, by typing (at the shell prompt):

```
lkb &
```

This will start emacs, our editor of choice, with the Lisp and LKB running as sub-processes to the editor. The benefits of this set-up will become apparent sooner or later.

Several new windows appear: the LKB Top window and the main emacs window, where we will be editing the files of our grammar. Starting with this exercise, we will introduce basic grammar engineering techniques and LKB functionality; we will initially take a playful approach, i.e. individual experimentation (trial and error; no worries, there is little risk of permanent damage to anything). Over the next few weeks, we will introduce more of the formal background, take into use more and more more of the LKB functionality, and we will successively improve and enlarge a small grammar of English.

(c) To obtain a starting package for this exercise, execute the following command from the shell prompt:

```
exercise2b
```

When executed for the first time, this command will create a new directory '`exercise2b/`' in your home directory, containing a set of files that comprise our statements about the grammar of English, using a specialized formalism called Type Description Language (TDL); we will have more to say about TDL in the lectures to come. Throughout this exercise, we will be making changes to the files in '`exercise2b/`'.

(b) Load the grammar by selecting Load | Complete grammar in the window called LKB Top, then double-clicking on the directory '`exercise2b`' and on the file '`script`'. Reassuring messages will appear in the LKB Top window, and a new window will pop up showing you the so-called type hierarchy for this small grammar.

With the mouse in the LKB Top window, select Parse | Parse input... from the menu. Type in the sentence *the cat chased the dog*, thereby replacing the existing contents of the new window that pops up. Click on the button OK. The system will parse the sentence and pop up a window containing a little parse tree for the single analysis of this sentence. Click on the parse tree to get a menu which allows you to enlarge it and look at the nodes in more detail.

2 Try the Simple Batch Parsing Mechanism (0 Points)

(a) In the LKB Top window, select the menu item Parse | Batch parse... which will pop up a window asking you for an input file to be processed.

Click on the file '`test.items`' in your grammar directory, then hit the button OK. This will pop up a new window asking you for the name of the output file where the results of the batch run will be stored; choose '`test.results`' for the output file and confirm that, indeed, you want this file to be overwritten.

The system will print the message *Parsing test file* in the `*common-lisp*` buffer in emacs when it starts, and will print the message *Finished test file* when it is done.

Open the file '`test.results`' in emacs and inspect the parsing results. The first number following each sentence is the number of distinct analyses (parses) found by the LKB parser; the second number is the total count of edges in the parse chart, i.e. a measure of the degree of ambiguity while parsing the sentence.

3 Add a Lexical Entry for Another Animal Noun (5 Points)

(a) In emacs, open the file '`lexicon.tdl`' for editing. Copy the five lines that define the lexical entry for *cat* and modify your copy to make the value of `ORTH` appropriate for another animal; also, assign a new identifier (the name preceding '`:=`') to your new lexical entry.

Save the changed version of the file. Reload the grammar and test the effect of your addition. In the LKB Top window, execute Load | Reload grammar. Study the messages printed to the LKB Top window; in case there are errors, correct your changes to '`lexicon.tdl`' and reload. Next, parse the sentence *the cat chased the animal* (substituting the name of your animal, of course).

(b) Add this sentence to the '`test.items`' file and rerun the batch check.

4 Poke Around the Grammar a Little (10 Points)

(a) Investigate the grammar in order to get an intuitive idea of how it works; we will discuss more formal details later. In particular, look at the following sentences and try and decide why they do or do not parse. For each of the above, summarize your findings in a sentence or two, preferably by creating a new file '`README`' inside the '`exercise2b/`' directory, such that your comments will be part of your final submission to us.

the cat barks

the cat chased

cat barks

the cat bark

bark

Note that the start symbol of our grammar (i.e. the equivalent of the category 'S' in a context-free grammar) is defined in the separate file '`roots.tdl`'. For a parse result to be accepted as 'sentential' its feature structure has to unify with this start symbol.

Note that the Parse | Show parse chart menu entry can give you an idea of which constituents were built, even when an input is not recognized by our grammar. Just like in the (enlarged) tree view, entries in the parse chart are mouse sensitive and allow inspection of the feature structure associated with each constituent. Notice that the grammar is parsing some sentences incorrectly (i.e. overgenerates) and failing to parse some sentences that should parse (i.e. undergenerates).

5 Adding a More Interesting Lexical Entry (10 Points)

(a) The rule that is needed for ditransitives (i.e. verbs that take two objects to their right) is in the grammar, but there are no lexical entries that utilize it. Add an entry for *gave* which takes two noun phrase complements (i.e. what is needed to parse, say, *that dog gave the cat the animal*).

Copy the entry for *chased* in '`lexicon.tdl`'. Replace the orthography value as before and assign a new lexical identifier to this entry (e.g. '`gave`'). Add an extra element to the `COMPS` list, which will be a duplicate of the one that is already there. Note that lists are delimited by angle brackets ('`<`' and '`>`'), and the elements on lists are separated by commas.

(b) Test by parsing *that dog gave the cat the animal*. Also test for overgeneration by confirming that you cannot parse *that dog gave the cat*. Add an appropriate set of test sentences to '`test.items`'.

6 Prepositional Phrases and PP Complements (15 Points)

(a) In order to parse *that dog gave the cat to the animal*, we will have to extend our grammar futher. Add the type *prep* as a new subtype of the type *pos* to the file '`types.tdl`', by copying the type description for *noun* (and replacing *noun* with *prep*).

Add a lexical entry for the preposition *to*. This should be similar to the entry for *chased* in that *to* will take a single noun phrase complement, but the value for `HEAD` should be *prep* and the value of `SPR` should be the empty list (i.e. '`<>`').

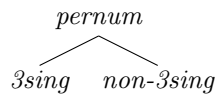
(b) Add a second lexical entry for *gave*. You can copy your existing entry but you will need to use a different identifier (i.e. the thing to the left of the '`:=`' operator), for example '`gave_np_pp`'. You also need to change the second element in `COMPS` to make this entry require a PP (we will not bother about making sure it is a PP headed by *to* yet).

(c) Add several test items, both grammatical and ungrammatical, to your '`test.items`' file, which will allow you to check the correctness of your additions to the grammar. Run the batch parsing utility again, and examine the results. Celebrate as appropriate.

7 Subject – Verb and Determiner – Noun Agreement (15 Points)

(a) Extend the grammar to capture subject – verb agreement, admitting e.g. *the dog barks* but not **the dogs barks*. We will introduce constraints on the `SPR` attribute of lexical entries requiring that person and number properties match between head and specifier. Rather than using separate features for number and person, we will use types that combine both properties, allowing a more direct encoding of English inflectional morphology.

(b) Add this small type hierarchy to the file '`types.tdl`', making *pernum* a subtype of *feat-struc*:



Also in '`types.tdl`', add the feature `AGR` to the type *pos*, with its value constrained to be of the new type *pernum* that you just added.

(c) In the lexicon file, add the appropriate constraint to each verb by restricting the `AGR` value inside of its `SPR`. Also in the lexicon, add the correct `AGR` value to each noun.

Save your changes, then reload the grammar, apply the batch test with the file '`agr.items`', examine your results, and make any necessary corrections (but ignoring determiner – noun agreement, for now).

(d) Extend your analysis to cope with determiner – noun agreement, admitting e.g. *these dogs bark* but not *these dog barks*. In the lexicon again, modify each noun's lexical entry by adding the appropriate constraint on the `AGR` value of its `SPR`. Also add the correct `AGR` value to each determiner.

(e) Check your revised grammar again using the file '`agr.items`', and make any necessary corrections. Add some additional test examples to this file with varied combinations of mismatch in agreement among determiners, nouns, and verbs. Then run the batch test and examine the results.

8 Use of Feature Structure Re-Entrancies in Agreement (5 Points)

(a) 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*.

(b) 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 still unclear, here is 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`.

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

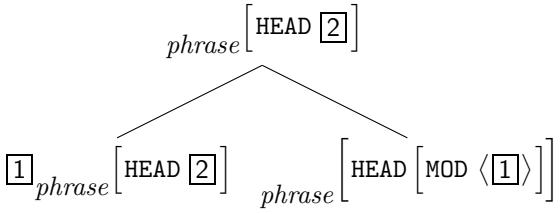
(a) So far, we have grammar rules to combine heads with two general types of sister constituents, viz. complements to the right of the head and specifiers to its left. Extend the grammar to provide an analysis of modification, admitting sentences like *the dog barks near the cat.*

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.

(b) Add *near* as an additional preposition in the lexicon, copying the entry of *to* and adapting it as needed.

(c) 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.

(d) 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.

(e) 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.

10 The Head Feature Principle (5 Points)

(a) 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 two new types:

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

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

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). The *head-initial* type, on the other hand, uses the TDL notation '...' to leave the rest of the ARGS list underdetermined, i.e. it could be empty or a remaining list with any number of elements.

- 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 occurence in the head – modifier rule perhaps.

11 Eliminating Redundancy in the Lexicon (15 Points)

(a) 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 which are true for all instances of each word class, e.g.

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

Introduce subtypes of the *noun-word* type for singular and plural nouns, and do the same for determiners.

(b) 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.

(c) 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" ].
```

12 Submitting Your Results

To provide your results to us, we will need to receive the entire contents of your '`exercise2b/`' directory when you are done, but no later than the end of the day on Sunday, April 3. For the IFI Linux environment, we provide a command-line tool for you to automate the process of submitting results to us; at the shell prompt, in the parent directory to '`exercise2b/`', try the following

```
submit exercise2b
```

This command should generate a screenful of reassuring messages about packaging up and delivering your files. When in doubt, contact Lars-Erik and Stephan in email. In case you would rather pack up your '`exercise2b/`' directory yourself, please email a compressed archive (e.g. in '`.tgz`' or '`.zip`' format) to both Lars-Erik and Stephan before the submission deadline.