Grammar Engineering — ESSLLI 2016 (Exercise 5)

High-Level Goals

- Perform list concatenation using open-ended lists and unification.
- Extend the grammar to include long-distance dependencies.
- Explore an analysis of relative clauses (when you get back home).

1 Grammar for Today

(a) Bring up our development environment by opening a terminal and running the '`lkb`' command. To put everyone onto a level playing field, please obtain a fresh starting package:

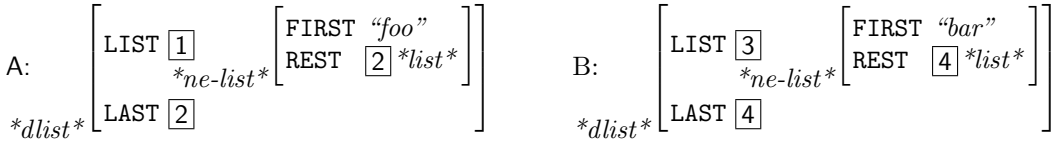
```
svn co $grammar5
```

(b) Before you will be able to parse anything using this grammar, we need to make a small number of changes, however. Move the attribute `ORTH` from *lex-item* to *expression* and change its value from **string** to **dlist**. In the lexicon, convert all values of the feature `ORTH` to one-element difference lists containing the original string, for example change `ORTH "dog"` to `ORTH <! "dog" !>`. Apply a similar change to the type *prep-lxm*, i.e. make sure that what is equated with the `FORM` feature is the (first) element of the `ORTH` value.

Make sure you can still load and run the resulting grammar before you move on.

2 Open-Ended Lists and Unification-Based Concatenation

- The typed feature structure formalism that is implemented in the LKB (and similar systems) excludes relational constraints (like `append()` or `reverse()` of lists). However, the *difference list* encoding in feature structures allows us to achieve list concatenation using pure unification. A difference list is an open-ended list that is embedded into a container structure providing a 'pointer' to the end of the list, e.g.



Now, using the `LAST` pointer of difference list A we can append A and B by (i) unifying the front of B (i.e. the value of its `LIST` feature) into the tail of A (its `LAST` value) and (ii) using the tail of difference list B as the new tail for the result of the concatenation (see Copestake, 2002, for a more elaborate discussion).

The goal of this exercise is to pass up the `ORTH` values from words to phrases and use list concatenation to determine the value of `ORTH` at each phrase. The top ('S') node in a complete analysis of a sentence should have as its `ORTH` value a difference list that contains all words of the sentence, preferably in the right order.

(a) Each rule will be required to concatenate the `ORTH` values of all daughters to the rule and make the resulting list the `ORTH` value on the mother. To avoid duplication of the append operation in the '`rules.tdl`' file, introduce types *unary-rule* and *binary-rule* that inherit from *phrase* and perform the concatenation of `ORTH` values.

(b) In order to have one specific type for each actual rule in the rules file and to allow unification of various *phrase* subtypes to succeed, we need to cross-multiply the two types accounting for arity (number of daughters) with the existing dimension we already have for phrases, namely the position of the head daughter (*head-initial* vs. *head-final*). Use multiple inheritance to create the following types combining the arity dimension with the dimension of head position: *unary-head-initial*, *binary-head-initial*, and *binary-head-final*.

(c) We also need to account for the effect on orthography of inflectional rules, which are subtypes of *word*. Since inflectional rules can change the `ORTH` value, we cannot simply identify the `ORTH` of a word with the `ORTH` of its `ARGS.FIRST` (the lexeme being inflected). But we still have to ensure that the `ORTH` difference list is terminated. So add a constraint to the type *word* making its `ORTH` value be a one-element difference list, which we can define as '`<! *string* !>`'.

(d) Rework the file '`rules.tdl`' to use the new, more specific rule types, as appropriate. Reload the grammar and check correctness by parsing a few sentences interactively and verifying that the `ORTH` value on 'S' nodes contains all the words that contribute to the sentence.

(e) Run the 'Batch parse' machinery on the '`all.items`' file and validate the results.

3 Unbounded Dependencies: Topicalization

Like many languages, English has constructions where at least one of the syntactic arguments of a predicate does not show up right beside its predicate, but somewhere farther away. One class of these constructions is called *long-distance dependencies*, illustrated by the following examples:

(1) *That cat, the dog wanted me to give to the aardvark.*

(2) *Examples like this, I'm sure nobody ever really says.*

Long-distance dependencies are discussed in more detail in Chapter 14 of Sag, Wasow, & Bender (2003); for the current exercise, it will be enough to remember that a sentence-initial phrase like *that cat* can be analyzed as providing the *filler* for a *gap* (a missing argument) somewhere later in the sentence.

- Prepare the basic machinery for handling long-distance dependencies. Our approach will be to remove a complement from the `COMPS` list of a predicate using a new syntactic rule, store it for awhile in a `GAP` attribute, and then retrieve it at the top of the phrase structure tree using a new syntactic rule.
- First, in the file '`types.tdl`' add the feature `GAP` to *expression*, and make its value be of type **dlist**, for reasons that will become clear in a moment. Then consider the various sub-types of *expressions*; we will assume that all lexical items have an empty `GAP` list. Make this so.
- Phrases may contain a gap, and that gap might come from any of the daughters of a phrase, so we use the same difference-list mechanism to collect up these values that we used for `ORTH` earlier already. Add a subtype of the type *unary-rule* called *unary-rule-pg* (short for 'pass gap') which makes the phrase's `GAP` be unified with the `GAP` of its only daughter, and add a subtype of the type *binary-rule* called *binary-rule-pg* which appends the `GAP` values of its two daughters. Extend the hierarchy below *binary-rule* to inherit from these new types.
- Now move to the file '`rules.tdl`' and modify the existing rules to inherit from the new 'gap passing' types.
- Still in '`rules.tdl`', add a new unary rule callled *startgap-rule* which extracts one complement of its daughter, moving it into the `GAP` list of the mother. Since this is where we introduce the missing argument, and we assume that only one gap per sentence is needed, the daughter in this construction should have an empty `GAP` value.
- Finally, add a new rule called *filler-head-rule* of type *binary-head-final* whose first daughter is identified with the single value of the `GAP` attribute on the second daughter. This is the rule that will combine, for example, *that cat* with *the dog chased* to build *that cat the dog chased*. Save your files, then reload the grammar, and check that you can parse the example *that cat the dog chased*. If not, make the required repairs, until you can parse this example.
- Now consider the examples *that dog the cat gave to the aardvark* and *to that cat the dog gave the aardvark*. Once these are parsing, try testing your analysis on the batch file '`gap.items`'. Use the earlier test items file to make sure a that there is no 'leakage' of gaps.

4 Relative Clauses (Optional)

- Now that we have an analysis of gaps, we can introduce significant additional coverage to the grammar by adding an analysis of relative clauses, as in *the dog that the cat chased barks*. Some of the modifications to the grammar will look similar to what you did for gaps, and some will be new. Our analysis will assume that relative pronouns like *that* introduce a feature which propagates up to the clause containing the relative pronoun, and that there is a separate unary rule which discharges this feature and constructs a phrase that is a postmodifier of nouns.

  You will have considerable freedom in implementing this analysis, but here are some initial steps and a few guidelines:

- Add the feature `REL` to *expression*, and make its value of type **dlist** for reasons similar to those for the feature `GAP`.

- Make sure that all existing syntactic rules (both *unary-rule* and *binary-rule* subtypes) preserve the values of `REL` from their daughters.

- Add a new lexical entry for the relative pronoun *that* to the lexicon, with most of the properties of an ordinary pronoun, but with its `REL` value a one-element list whose single element is reentrant with the expression in `MOD`. You may want to consider making this entry an instance of *word* rather than *lexeme*.

- Make sure that all other lexical entries have an empty value for `REL`, modifying the lexical type hierarchy appropriately.

- Add the unusual non-branching rule that converts a sentence containing a relative pronoun into a noun-modifying phrase. Notice that this rule will have a `HEAD` value on the mother that is not identified with the `HEAD` value of its (only) daughter, since verbs have an empty `MOD` value, but we want to build a phrase whose `MOD` value is non-empty.

- Not all relative clauses in English contain the relative pronoun *that*, of course, and in fact, some relative clauses contain no pronoun at all, as in *the dog the cat chased barks*. Extend your grammar to provide an analysis for sentences containing these so-called 'that-less' relative clauses. As always, include a file of example items to showcase the coverage of your analysis.