

Formal Syntax & Grammar Engineering (Exercise 5)

High-Level Goals

- Perform list concatenation using open-ended lists and unification.
- Finally, add semantics to lexical entries and grammar rules.
- Use the LKB generator to identify overgeneration in the grammar.

1 Obtaining the Starting Grammar

- Besides good linguistic and computational judgment, grammar engineers also often require some sophistication in file manipulation and merging. As the starting package for this *final* exercise is the result of the preceding assignment, we suggest you take the following preparatory steps.
- (a) Make sure you have completed at least part (3) of the fourth exercise, i.e. have a grammar that loads without errors, parses the right set of sentences, and comprises its lexicon exclusively of lexemes. Create a copy of your ‘grammar3’ directory, such that you will be able to fully complete the preceding exercise and submit your results later today:

```
cd ~
cp -pr grammar3 grammar4
```

Next, within the new ‘grammar4’ directory, obtain a few new files we supply for this exercise:

```
cd grammar4
cvs update
```

- (b) Include the contents of the file ‘extras.tdl’ at the end of your own ‘types.tdl’ file. 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 PFORM 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.

$$\text{A: } \left[\begin{array}{l} \text{LIST } \boxed{1} \\ \text{REST } \boxed{2} *list* \\ \text{LAST } \boxed{2} \end{array} \right] \quad \text{B: } \left[\begin{array}{l} \text{LIST } \boxed{3} \\ \text{REST } \boxed{4} *list* \\ \text{LAST } \boxed{4} \end{array} \right]$$

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 `<! [] !>`.
- (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 Adding Semantics

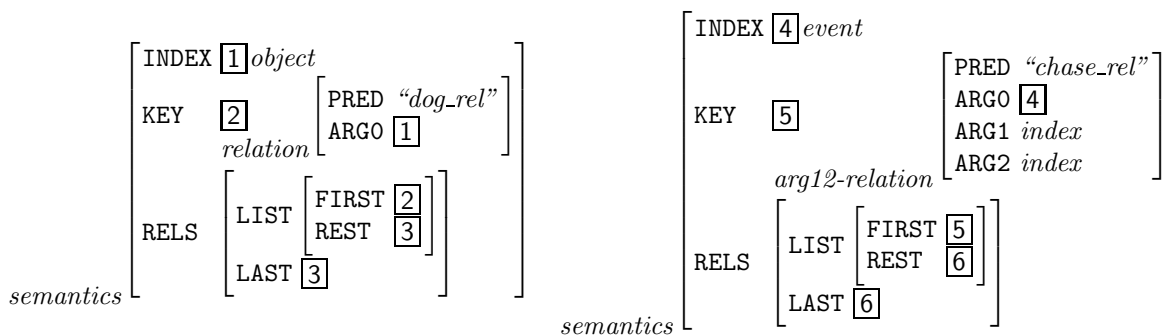
- Our approach to adding semantics to the grammar builds on list concatenation as the basic operation of composition. Semantic relations are introduced by lexical entries and successively combined as words are combined with other words (or phrases) to form larger phrases. We will use a type *relation* to capture the basic units of semantics that are associated with words, and we'll use subtypes *arg1-relation*, *arg12-relation*, and *arg123-relation* for predicates of corresponding arity:

$$\begin{array}{c}
 \textit{relation} \left[\begin{array}{l} \text{PRED } *string* \\ \text{ARGO } index \end{array} \right], \quad
 \textit{arg1-relation} \left[\begin{array}{l} \text{PRED } *string* \\ \text{ARGO } index \\ \text{ARG1 } index \end{array} \right], \quad \dots \quad
 \textit{arg123-relation} \left[\begin{array}{l} \text{PRED } *string* \\ \text{ARGO } index \\ \text{ARG1 } index \\ \text{ARG2 } index \\ \text{ARG3 } index \end{array} \right]
 \end{array}$$

- (a) Add an atomic type *index* with subtypes *object* and *event*, which we will use to represent variables in role assignment within relations.
 - (b) Add the types *relation* through *arg123-relation* (as shown above) below *feat-struct*.
- To associate semantics with words and phrases, we need another type that will serve as the value of a new `SEM` attribute in *expression*:

$$\textit{semantics} \left[\begin{array}{l} \text{INDEX } index \\ \text{KEY } relation \\ \text{RELS } *dlist* \end{array} \right]$$

Roughly speaking, the `INDEX` attribute corresponds to the external variable that is available for binding, the `KEY` attribute points to the distinguished relation that is used for semantic selection (typically contributed by the semantic head; see the MRS paper on the course web site, if you strongly want to learn more about our specific approach to semantics), and the `RELS` attribute holds a list of relations (see below). For the lexical entries 'dog' and 'chase', respectively, we want the following semantics (as the value of their `SEM` feature):



- (c) Introduce the type *semantics*, add the feature **SEM** to *expression*, and constrain its value to *semantics*.
- (d) Enrich the *lexeme* type to reflect that (i) lexical items have a singleton **RELS** list, (ii) the **KEY** relation corresponds to the first (and only) element in **RELS**, and (iii) the **INDEX** is the **ARGO** of the **KEY**.
- (e) Enrich the types *det-lexm*, *noun-lexm*, and *verb-lexm* (or their equivalents) to constrain the semantic **INDEX** to be of type *object* (for determiners and nouns) and *event* (for verbs), respectively.
- (f) Add a unique semantic predicate name, as the value of **SEM.KEY.PRED**, to each entry in the lexicon. Reload the grammar and use the ‘View – Lex Entry’ menu command to inspect the lexical entries for ‘dog’ and ‘chase’, making sure they look as specified above (ignoring for now the linking of **ARG1** et al. values to syntactic arguments, of course, which will also affect the specificity of relation types you will see for now).
- Roughly similar to **ORTH**, each phrase will accumulate the semantic contributions from each of its daughters and use list concatenation in building up the **RELS** value; while technically we use a list for this purpose, the order of elements in **RELS** is actually irrelevant: we are using a list to represent a bag (or multi-set) of objects. Add the principles of semantic composition.
- (g) While unary rules simply pass up the entire **SEM** value from the daughter to the mother, binary rules apply difference list concatenation to the **SEM.RELS** values.
- (h) In each phrase, the **INDEX** and **KEY** values are contributed by the semantic head, which can in principle be distinct from the syntactic head daughter. For this exercise, however, we’ll always take the syntactic head to be the semantic head. Add appropriate co-indexation of **INDEX** and **KEY** to the *head-initial* and *head-final* types to express this generalization.
- (i) Inflectional rules are subtypes of *word* but behave much like unary rules; add the passing up of the **SEM** value from the daughter to that of the mother on the type *word*. Also modify the derivational rule for dative-shift (a lexeme-to-lexeme rule) to preserve the semantics from daughter to mother. Note that, in general, derivational rules would be expected to alter the semantics, like the agentive rule which we have excluded for this exercise.
- (j) Reload the grammar, eliminate remaining errors (if any), and make sure that (i) coverage remains unchanged and (ii) the **RELS** value on ‘S’ nodes contains the relations of all words in the input sentence. Since each word contributes exactly one relation in our grammar, the number of elements in the **RELS** list should always be the same as the number of words in the sentence.
- What remains to be achieved is the linking of syntactic arguments to semantic roles. We will use the **INDEX** value of arguments (often called the *index*) to make this linking explicit. This requires enriching the types for all lexemes that take arguments (i.e. have non-empty **SPR** or **COMPS** lists, and likewise for non-empty **MOD** values on modifiers) to identify the **INDEX** value of each argument with exactly one role in the relation of the functor (the semantic head).
- (k) Nouns identify the **INDEX** of their specifier with their own **INDEX** (also their **ARGO**).
- (l) All verbs identify the **INDEX** of their specifier with their **ARG1** role; in addition, transitive verbs identify the **INDEX** of their complement with their **ARG2**; analogously, ditransitive verbs map their complements to **ARG2** and **ARG3** (note, however, that for the dative alternation we expect identical meanings).

- (m) Prepositions are special: they identify the INDEX of the *expression* selected for by MOD with their own INDEX, and map the INDEX of their complement to ARG1. Adjectives are similar to prepositions, but have no ARG1 role.
- (n) Reload the grammar, confirm everything works, and admire the beauty of semantic composition. Make sure to verify that in sentences like ‘the dog barks’, ‘the cat gave that dog those aardvarks’, or ‘the cat barked near those dogs’ the following well-formedness conditions hold in the semantics: (i) all INDEX values are specialized to either *event* or *object*, (ii) determiner and noun in each noun phrase share the same ARGO variable, (iii) each role in a verbal relation is bound to the index of the corresponding argument, (iv) the ARGO of a prepositional phrase *modifier* is bound to the ARGO of the noun or verb being modified, and (v) the ARGO of adjectives is bound to the ARGO of the noun it modifies.
- (o) The LKB provides facilities to read out a semantic formula from a feature structure, print it in various formats, perform (limited) semantic inference, and generate from it. Because our semantics is (currently) very shallow, however, only a subset of this functionality can be used meaningfully.

From the tree summary display (the window showing tiny trees after parsing), execute the menu commands ‘MRS’ and ‘Indexed MRS’ to see a readable form of the semantics produced by your grammar.

4 Generation: The Inverse of Parsing

- The LKB comprises a chart-based generator that, given a suitable semantic formula, can generate all sentences (i.e. well-formed strings) that correspond to this semantics. For the generator to work, we need to make a final small extension to our grammar:
 - (a) For generator-internal purposes, the generator requires INDEX values to have one appropriate magic feature, SKOLEM, whose value is of type **string**. Enrich the type definition for *index* appropriately. Reload the grammar.
 - (b) To get full access to the LKB generation component, execute the ‘Options – Expand Menu’ command. Then, to make the lexicon accessible to the generator, execute ‘Generator – Index’; if there are warnings or errors at this stage, revisit your grammar. In order for the indexing to happen at load time, edit the file ‘script’ in your grammar directory which is the load file for the grammar. Towards the end of the file, uncomment the line (`index-for-generator`) by removing all semicolons.
- Once indexing is complete, from the tree summary display explore the output of the ‘Generate’ command for various sentences; this is a short-cut command for extracting the indexed semantics from a feature structure and using that as the input for the generator.

5 Eliminate More Overgeneration (Optional)

- At this point, generating from the semantics of *the aardvark barks* should return four sentences. Investigate the reasons for this overgeneration, identify the missing bits of information, and speculate about changes to the grammar that would be needed to eliminate this ambiguity. Submit a few paragraphs of text summarizing your findings.
- Use the LKB to (re-)generate from the semantics of *that cat gave those dogs the aardvark* and *that cat gave the aardark to those dogs*. Inspect the associated semantics and determine why the two sentences are *not* considered semantically equivalent yet. In one paragraph (or two) of text, sketch a modification to the grammar that will make both usages of ditransitive verbs like *give* result in equivalent semantics. What is the main challenge here?

Submit your results in email to Stephan and Lilja by 18:00 h on Tuesday, December 14.