Topics in Computational Linguistics
Parsing & Generation (Exercise 6)

Goals

1. Put everything together: make our active chart parser use the unifier;
2. reduce the cost of feature structure manipulation; avoid unnecessary unifications.

Obtain the Starting Package and Bring up the LKB

(a) As always, bring up bash, obtain a starting package by typing `cvs checkout week6` (in bash), launch emacs, and start the LKB.

(b) Open the files `dag.lsp` and `active.lsp`, where we will be making changes throughout this exercise.

1 Unification-Based Parsing (20 + 5 + 15 Points)

For a change, we have done quite a bit of work for you. The global structure `*grammar*` now has the following components:

- **types** — list of all types defined in the grammar, each a type structure with components `id` and `dag`;
- **lexemes** — list of lexical entries, lexeme structures with components `id`, `dag`, and `form`; we will use the `form` value (a symbol) to look up lexical entries and `dag` as the category on initial lexical edges;
- **rules** — list of grammar rules, rule structures with components `id`, `dag`, and `rhs`; the `dag` value is the full feature structure, including daughters in its `ARGS` list, and `rhs` a list of paths into the structure;
- **roots** — list of grammar start symbols, each a root structure with components `id` and `dag`.

We have modified the parse chart initialization in `active.lsp` to take advantage of these new structures; go have a look at the revised `parse()` function to see how the new `*grammar*` organization works. In the unifier, we have augmented `unify()` to take an optional `path` argument, requesting that the second `dag` be unified into the first at the given path. Also, our starting version of `dag.lsp` already supplies a predicate `equivalentp()` to compare two dags for structural equivalence (all types and arcs on all nodes are the same), though this may require a little more work later in this exercise.

(a) Complete the conversion of the chart parser from dealing with atomic categories to feature structures. Identify the few remaining parts of the code in `active.lsp` that deploy category equality when comparing edges and make the necessary changes to deal with feature structure categories. Consider writing a function `non-destructively-unify()` that takes three arguments—two dags plus a path pointing to an argument position in the first dag—and returns the result of the unification without permanently altering the input dags.

(b) In selecting the parsing result(s) from the chart, `parse()` calls a function `sentencep()`, which we need to implement for edges with feature structure categories. `sentencep()` takes an edge as its argument and returns true if the edge is (i) passive and (ii) compatible (aka unifiable) with one of the start symbols of the grammar.

(c) At this point, you should be able to test the parser on sentences from our new domain (animals in potentially violent action). Test at least a few examples, non-ambiguous and ambiguous, and compare the results you get to the LKB built-in parser (available through the `Parse – Parse input` menu from the LKB top window). You will probably find that the pretty printing of edges is not quite ideal (with `nil` showing as the edge category), but we will defer this issue briefly. To inspect the feature structures associated with edges, note that you can invoke the `browse()` function on all objects with DAGs, i.e. types, rules, lexemes, start symbols, edges, and (of course) dags themselves.

(d) To enable more readable printing of edge structures (and for later optimization of the parser), we will need to allow each edge to keep track of the structure from the grammar that gave rise to it, i.e. the lexeme or rule that ‘licensed’ an edge. Our edge structure (as of this week) already has a slot `sign`, which we will have to set appropriately in all `make-edge()` calls; e.g.
2 Feature Structure Manipulation Costs (5 + 10 + 5 Points)

Another attractive element of our starter kit for this week is an abstract data type to keep track of feature structure traversals (our basic measure of the complexity of feature structure manipulation). In `dag.lsp` you will find a new global variable *counter* and functions count-copy(), count-failure(), count-success(), count-equivalence(), and count-traversal() to record top-level calls to copy(), unify() (successful or non-successful), and equivalentp(), respectively; in addition to these fine-grained records, the global traversal counter should be the sum of all feature structure traversals, i.e. one unification will be counted as one traversal and as a success or failure, depending on the unification result.

(a) Go through the dag manipulation functions in `dag.lsp` and add call counting accounting, as appropriate. Note that restore() calls itself recursively, but we only want to count the top-level calls; add an optional argument recursivep to the function definition (that defaults to nil) and arrange for recursive calls to not increment the traversal counter. In the main parse() function, add a call to counter-initialize() to make sure that the counting starts afresh for each parse.

(b) At this point, you should be able to reproduce the following results:

? (parse '(the dog barks))
→ (#E[28: (0-3) head-specifier-rule --> 26 18 .])
? *counter*
→ #C[98 copies + 36 failures + 13 successes + 3 equivalences = 254 traversals]

Explain in a few sentences in what sense these numbers actually do add up. Experiment with longer and more ambiguous sentences to see the effects of a larger search space explored by the parser.

(c) To reduce the cost of feature structure manipulation, we will implement the generation counting scheme introduced on slide # 63. Make the required changes to the dag abstract data type, save and reload, and study the effects. In case you see that little has changed to earlier results, remember that the (relatively expensive) restore() function is expected to benefit most directly from the refined dag type. Again in a few sentences, comment on the effects of the new scheme.

3 Filtering Impossible Unifications (20 + 10 Points)

The call counting clearly shows that a large number of unifications fail. Given off-line inspection of the grammar rules, we should be able to predict a large proportion of those unification failures without even having to call unify(). In this part of the exercise, we will compile what is often called a rule filter, conceptually a table indexed by rules and their argument positions providing the information about impossible feeding relations. For example, prior to parsing we can determine that the specifier – head rule can never feed into the first argument position of the head – complement rule, simply by inspection of the two feature structures.

(a) For maximum efficiency, we will implement the rule filter as an additional element of the rule structure. In our usual supportive way (for this week), we have already supplied a component filter in the definition of rule. The purpose of this exercise is to cross-multiply all rules in all argument positions and compute, for each rule and each argument position, the list of grammar rules that cannot unify into that argument position. In other words, we want the filter value on a rule to be a list of sets: the top-level list will contain one set per argument position (i.e. there will be two list elements for a binary rule), and each set is the collection of rules that, by static inspection of the grammar, was found to be incompatible with that argument position. Look at the definition of compatiblep() towards the end of 'active.lsp' to see how the rule filter will be put to work. compatiblep(), when called with one active and one passive edge, retrieves the filter value on the rule underlying the active edge (as this rule initially determined the
argument positions of the edge) and indexes the filter according to the numerical index of the argument position to be filled next, i.e. the difference between the total number of argument positions and the remaining unanalyzed ones.

Implement the body of `compute-rule-filter()` (in `active.lsp`) to augment each rule in `*grammar*` with a suitable list (of sets of incompatible rules) as its filter value. Remember that you can set the variable `*pretty-print-structures-p*` to `nil` to disable pretty printing of objects (e.g. rules while debugging this part of the exercise) and inspect the ‘raw’ content of each rule. While pretty printing is enabled, you will only note the effects of `compute-rule-filter()` indirectly, as the number of rules (over all argument positions) that were found incompatible is included in parenthesis in the print-out for `rule` objects, e.g.

? (third (grammar-rules *grammar*))
→ #R[head-complement-rule-0: #D[head-initial ...] --> (ARGS FIRST) (5)]

(b) Once you are content with your implementation of the rule filter computation, find the right place in the parser to add a call to `compatiblep();` then, compute the rule filter, and use the call counting mechanism to study the effects on parsing efficiency. If you are unable to produce the expected parsing results, your filter may be overly restrictive (filtering unifications that could have succeeded), if you fail to see a reduction in feature structure traversals, your filter may be defective. Make the necessary corrections until you observe a tangible reduction in unification failures; then consider inclusion of lexemes in the computation of the rule filter. Re-run your experiments and comment on the results.

4 Ambiguity Packing (10 + 0 Points)

Surely you must have noticed that our unification-based parser, at this point, fails to perform any packing of chart edges, i.e. we always return as many edges from the `parse()` function as we expect analyses for the input string. However—since we have replaced the original `equal()` comparison of atomic categories with a call to `equivalentp()` when looking for a suitable host in `pack-edge()`—we should expect the ambiguity for those dogs chased the cat near that aardvark, say, to pack nicely into a single edge.

(a) Use the `browse()` function on both edges returned by your parser for this input (or use the built-in LKB parser and view the feature structures of the top nodes on the two trees; or use the LKB chart browser and inspect the candidate edges for ambiguity packing): start from the assumption that the implementation of `equivalentp()` that we have kindly supplied is correct (or perform a static code review of our function to convince yourself) and compare the feature structures to work out in which sense they are, indeed, not equivalent. This analysis will lead you to an important, new or familiar, insight into the nature of the encoding we have adopted for unification-based grammar rules; summarize your findings in a few sentences. Why is it impossible for the parser in its current form to perform local ambiguity packing?

(b) Modify the definition of `equivalentp()` to address this problem; test on a few sentences of increasing ambiguity and note the benefits of ambiguity packing.

Submitting Your Results

Since our programs are of some interesting complexity at this point, please make sure to submit the contents of the entire ‘week6’ directory.

Submit your results in email to Dan and Stephan by noon on Tuesday, May 20.