Computational Linguistics II (Fall 2006, Exercise 6)

High-Level Goals

1. Solidify our understanding of Common-Lisp concepts: local variables and degrees of equality;
2. implement the destructive unifier and a structure-preserving copy function for feature structures;
3. finally, put everything together: make our active chart parser use the unifier.

Obtain the Starting Package and Take a Tour of the LKB

(a) As always, bring up bash, obtain a starting package by typing `cvs checkout grammar9` (in bash), launch emacs, and start the LKB. We have provided a simple unification grammar that will provide the basis for our own implementation of graph unification and unification-based bottom-up chart parsing. Use the *Parse – Parse input* menu to parse one or two sentences (e.g. *the dog barks*) and study the feature structures of our grammar; they should look terribly familiar.

(b) To view the type hierarchy underlying the grammar, select *View – Type hierarchy* and choose **top** to browse the complete hierarchy (everything below the most general type). Notice how types like *adv* and *noun-word-3sing* have multiple parents, i.e. inherit from more than one supertype. In testing our unifier, we will want to compare unification results we obtain to the feature structures recorded in the LKB, as we can browse them by selecting *Expanded type* on any of the nodes in the type hierarchy display.

(c) Open the file '`dag.lsp`' (from the '`grammar9`' directory). Once again, we will be fleshing out skeletal (dummy) functions provided in this file, typically by replacing occurrences of '`???`' in the function bodies with an actual implementation.

1 Identity vs. Equivalence of Objects (15 Points)

(a) Write a recursive function `isomorphicp()` that can be used to compare symbols, numbers, and lists of symbols and numbers. `isomorphicp()` takes two arguments and evaluates to true if these are equivalent (which includes token identity and value or structural equivalence), e.g.

```
? (isomorphicp 'foo 'foo) → t
? (isomorphicp 42 42.0) → t
? (isomorphicp '(42 foo) '(42.0 foo)) → t
? (isomorphicp '((42 (foo))) '((42.0 (foo)))) → t
```

Note: For the purpose of this exercise, we will assume that the built-in predicates `equal()` and `equalp()` are compromised and must not be used in the definition of `isomorphicp()`. Instead, the function definition will have to determine what *types* of arguments it is given and act *appropriately.* If you find it difficult to choose a comparison predicate for numbers, re-read our slide copies from earlier in the course (or some of the code of this exercise).

(b) Consider the following pairs of assignments to variables `foo` and `bar`; for each pair, determine the truth value of '`(eq foo bar)`' (preferably through introspection rather than experimentation) and explain the results:

```
? (defparameter foo '(1 2 3)) → foo
? (defparameter bar '(1 2 3)) → bar

? (setf foo '(1 2 3)) → (1 2 3)
? (setf bar foo) → (1 2 3)

? (setf foo (cons 0 foo)) → (0 1 2 3)
? (setf bar (cons 0 bar)) → (0 1 2 3)

? (setf foo (rest foo)) → (1 2 3)
? (setf bar (rest bar)) → (1 2 3)
```

2 Implementing the Unifier (30 Points)

(a) For the representation of typed feature structures, we will rely on two abstract data types—called *dag* and *arc*—to encode a node in a dag structure and a feature – value pair in a dag, respectively. In '`dag.lsp`', define the structure *dag*, with components *forward*, *type*, *arcs*, and *copy*, and also define *arc*, with components *feature* and *value*. While the first three components of *dag* should feel familiar from your reading of Wroblewski (1987), we will have more to say on the *copy* slot later in this exercise. Once you have definitions for the *dag* and *arc* structures, edit the file '`script`' to delete the comment character (the semicolon) at the start of the last line, save everything, and reload the grammar.

(b) Next, implement the function `deref()`, which recursively follows non-`nil` *forward* pointers until it reaches a dag that is not forwarded. Throughout all dag-manipulating functions, we need to make sure that dags are always dereferenced (or 'forward-resolved') at each level, so as to avoid looking at the *type* or *arcs* slots of a dag that has a non-`nil` *forward* value.

As we did for the *edge* structure and others before, we have already arranged for you for dags to print (more or less) readably. For debugging, however, you may choose to turn off pretty-printing of (all) structures to see the 'raw' underlying representations, e.g.

```
? (defparameter foo
    (make-dag :type 'word
              :arcs (list (make-arc :feature 'agr :value (make-dag :type '3sing)))))
? (setf *pretty-print-structures-p* nil)
? foo
```

(c) Go through the definition of `unify1()`, the main function of the unifier: `unify1()` can count on the top-level `unify()` function to establish a `catch()` context for non-local exits (as unification failure is detected at some arbitrarily deep recursion level). At this point, ignore the optional *path* parameter to the unifier; for the time being, we will always unify two top-level feature structures, but the parser will later want to unify one structure *into* another structure (e.g. *path* will take on values like '`(ARGS REST FIRST)`' then). In our typed feature structure universe, we will use `nil` to denote the inconsistent 'bottom' type of the type lattice, i.e. indicating failure of unification. In essence, `unify1()` implements a typed variant of the destructive unification algorithm proposed by Wroblewski (1987): after dereferencing both input dags, it first determines the unification (aka greatest lower bound) of the types on the two structures. The computation of the greatest lower bound for two types is available through the function `glb()` (which we supply; maybe experiment with the `glb()` interactively: call it with two symbols naming types from the hierarchy as its arguments). Only if a greatest lower bound exists, can unification proceed; `unify1()` then puts the two input dags into one equivalence class, records the (potentially new) type on the result, and then combines all attributes from both dags.

(d) Since our unifier is *destructive* (i.e. permanently changes both its input dags), it is essential to make sure not to modify any of the structures that are part of the grammar. To avoid doing damage to the grammar, we will typically create a copy before invoking a destructive operation (like `unify()`) on it. Copying, in a nutshell, walks through the dag, creating copies of each node and all arcs, such that the resulting dag is structurally equivalent to the original but shares no elements with it (i.e. no two dag nodes or arcs in the original and copy are token-identical).

Our dag representation of typed feature structures builds on token-identity (aka `eq()`-ness) of nodes to encode coreference (aka feature structure reentrancy): where two paths in a feature structure refer to the same value, the underlying dag structure has one node occurring as the value in multiple feature – value pairs. A full traversal of the structure will thus lead to multiple visits to that node. Use the LKB to inspect the definition of the type *head-initial* in our grammar. Informally, what it states is that in head-initial constituents the value of `HEAD` on the mother is reentrant with that of the first daughter (i.e. the first element in the `ARGS` list).

Next study the result of the following code fragment (or consider first writing a function `dag-arc-value()` which, given a dag and a feature, returns the value of that feature in the dag—`dag-arc-value()` would allow elimination of the frequent `loop()`s below) to be sure you understand how reentrancy works in our feature structures:

```
? (let* ((dag (type-dag (lookup-type 'head-initial)))
         (head (loop
                  for arc in (dag-arcs dag)
                  when (eq (arc-feature arc) 'head) return (arc-value arc)))
         (args (loop
                  for arc in (dag-arcs dag)
                  when (eq (arc-feature arc) 'args) return (arc-value arc)))
         (first (loop
                   for arc in (dag-arcs args)
                   when (eq (arc-feature arc) 'first) return (arc-value arc))))
    (eq head (loop
                for arc in (dag-arcs first)
                when (eq (arc-feature arc) 'head) return (arc-value arc))))
→ t
```

Keeping our representation of reentrancy in mind, the identity of nodes that appear under more than one path in a structure must be preserved when creating copies of dags; the function `copy1()` will use the *copy* slot of the *dag* structure to allow each node of the original dag to (temporarily) keep a reference to its corresponding copy. In other words, `copy1()` checks the *copy* slot for each node that it visits before creating a new dag: where the *copy* slot is empty, a fresh dag copy is created and recorded in the *copy* slot of the input dag; whenever `copy1()` finds itself visiting the same input node twice (indicating reentrancy), the copy made earlier will be available in the *copy* slot and can be reused. Reusing the same (copied) dag multiple times in the emerging copy of the input structure has the intended effect of preversing reentrancy.

Go through occurrences of '`???`' in the definition of `copy1()` in '`dag.lsp`' and fill in the missing parts.

(e) Next, we need a function to reset the *copy* slots of all dag nodes in a feature structure to empty values (i.e. `nil`), which we will use to undo the temporary effects of `copy1()` on the input dag after the completion of each copy operation (otherwise later copies might end up re-using dags that form part of an earlier copy). Implement the body of `restore()` in '`dag.lsp`'. Finally, to complete the copy procedure, provide the definition of the top-level entry point `copy()`, making sure that the input dag is restored after the auxiliary function `copy1()` has been called and returned the actual copy.

Note: You may have noticed that, unlike in earlier exercises, we have hardly encouraged you to do testing of individual functionality so far. Without the ability to copy both input structures prior to unification, `unify()` would destructively modify the internal dags of the grammar and what worked once may not work the next time. However, damage to the grammar internals may still result while we have not confirmed proper operation of the `copy()` procedure. While debugging the copier and unifier, be sure to reload the grammar frequently (from the LKB menu) in order to reinitialize all grammar-internal dag structures.

(f) As indicated earlier, the value of the global variable `*grammar*` has been reorganized to be a structured object holding the various parts of the grammar—types, rules, lexical entries, et al. For the purposes of this part of the exercise (validating the unifier) we will be exclusively concerned with the *types* component in the *grammar* structure. Each type corresponds to what you can inspect in the LKB *View – Expanded type* menu (or from the type hierarchy browser) and is implemented as a *type* structure with components *id* (the type name as a symbol) and *dag* (the feature structure of the type), including all information inherited from super-types.

To test the unifier, use the function `lookup-type()` (see above) to retrieve pairs of types, extract their *dag* values, copy them, and then invoke the unifier on them, e.g.

```
? (unify (copy (type-dag (lookup-type 'noun-word)))
         (copy (type-dag (lookup-type '3sing-word))))
```

Also, note that you can invoke the LKB feature structure viewer on a dag object by calling the `browse()` function on it.

(g) Finally, to get a somewhat more substantive test, complete the body of `test()` in '`dag.lsp`'. The purpose of the `test()` function is to iterate over all types of the grammar and attempt to unify them against all types of the grammar (including themselves). Whenver the unification succeeds, `test()` is to print a line like the following

```
'POSTMODIFIER' & 'PREMODIFIER' = 'ADV'
```

which we take to indicate that *postmodifier* and *premodifier* successfully unify to *adv* (consult the LKB type hierarchy to see why). A complete list of successful unifications for this grammar is in the file '`GLBS`'; once you have verified the implementation of your `test()` function and feel content with the results, compare the print-out you get to our file.

3 Variable Binding (15 Points)

(a) Consider the following definitions for two global variables and one function.

```
? (defparameter foo 4) → foo
? (defparameter bar 2) → bar

? (defun mystery (foo)
    (format t "~a~%" foo)
    (format t "~a~%" bar)
    (let ((foo 2)
          (bar (+ foo 10)))
      (format t "~a~%" foo)
      (format t "~a~%" bar)
      (let* ((foo 3)
             (bar (+ foo 10)))
        (format t "~a~%" foo)
        (format t "~a~%" bar))))
→ mystery
```

Explain the print-out that results from each of the `format()` calls in the body of `mystery()` when evaluating '`(mystery 42)`'. Maintain a table to keep track of the values for *foo* and *bar* at each step.

(b) The built-in Common-Lisp *special forms* `push()` and `pop()` manipulate lists with a stack semantics (last-in, first-out), e.g.

```
? (defparameter *stack* '(1 2 3)) → *stack*
? *stack* → (1 2 3)
? (push 0 *stack*) → (0 1 2 3)
? *stack* → (0 1 2 3)
? (pop *stack*) → 0
? *stack* → (1 2 3)
```

Experiment with `push()` and `pop()` and consider writing functions that have the exact same behaviour. Why is it *impossible* to (re-)implement `push()` and `pop()` as functions?

4 Unification-Based Parsing (40 Points)

For a change, we have done quite a bit of work for you when it comes to building a unification-based chart parser. The global structure `*grammar*` provides 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 (when compared to our earlier, context-free variant) to see how the new `*grammar*` organization works. In the unifier, our `unify()` routine is prepared to take an optional *path* argument, requesting that the second dag be unified *into* the first at the given path.

Also, our current 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 function 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 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. 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 for this course. 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.

```
? (let ((rule (first (grammar-rules *grammar*))))
    (make-edge :from 0 :to 42
               :category (rule-dag rule) :unanalyzed (rule-rhs rule) :sign rule))
→ #E[1: (0-42) head-modifier-rule --> . (ARGS FIRST) (ARGS REST FIRST)]
```

Make the necessary adaptations in the parser and confirm that *all* edges have correct values in the *sign* slot; demonstrate test cases for different types of edges.

5 Ambiguity Packing (Optional; 10 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 our 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.

6 Filtering Impossible Unifications (Optional; 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.

```
? (second (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.

7 Submitting Your Results

Since our programs are of some interesting complexity at this point, please make sure to submit the contents of the *entire* '`grammar9`' directory.

Submit your results in email to Dan and Stephan by noon on Saturday, November 18.