Topics in Computational Linguistics
Parsing & Generation (Exercise 5)

Goals

1. Solidify our understanding of new Common-Lisp concepts: local variables and degrees of equality;
2. implement the destructive unifier and a structure-preserving copy function for feature structures.

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 week5` (in bash), launch emacs, and start the LKB. As we move towards unification grammars for the remainder of the quarter, we have changed the grammar to use types and feature structures in representing lexical entries, rules, and generalizations over classes of linguistic entities (also, we have moved to a slightly more action-oriented domain for the language generated by the grammar). If you are not already familiar with the LKB and unification grammars, use the `Parse – Parse input` menu to parse one or two sentences (e.g. `the dog barks`) and study the structure of our new grammar. In the window showing the parse result, use the mouse to click on the tree and select `Show enlarged tree`; on the new tree, click on the top node and choose `Feature structure` to view the complete typed feature structure derived for the input sentence.

(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 `week5` 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 (10 + 5 Points)

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

```lisp
? (equivariantp 'foo 'foo) → t
? (equivariantp 42 42.0) → t
? (equivariantp '(42 foo) '(42.0 foo)) → t
? (equivariantp '((42 (foo))) '((42.0 (foo))) → t
```

Note: For the purpose of this exercise, we will assume that the built-in predicate `equal()` is compromised and must not be used in the definition of `equivariantp()`. 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 that came as part of a recent 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:

```lisp
? (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 (2 + 8 + 20 + 25 + 10 + 0 + 5 Points)

(a) For the representation of typed feature structures, we will rely on two abstract data types—`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 already be familiar from the first lecture on unification, 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 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 'noun
           :arcs (list (make-arc :feature 'agr :value (make-dag :type '3sing))))))
? (setf *pretty-print-structures-p* nil)
? foo
? (browse 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). 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). 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 preserving 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 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 week (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, remember 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). Whenever the unification succeeds, test() is to print a line like the following
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 (10 + 5 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?

Submitting Your Results

Since our programs are of some interesting complexity at this point, please make sure to submit the contents of the entire ‘week5’ directory.

Submit your results in email to Dan and Stephan by noon on Tuesday, May 13.