Algorithms for AI and NLP (Fall 2008, Exercise 4)

Goals

1. Understand fully the generalized chart parser; unpack complete trees from the parse forest;
2. practice reading off grammar rules from a treebank; estimate conditional PCFG probabilities;
3. implement the destructive unifier and a structure-preserving copy function for feature structures.

1 Bring up the Editor and the Allegro Common Lisp Environment

- As always, start our integrated Lisp development environment, by typing (at the shell prompt):

  ```
  acl &
  ```

- For this session, we provide several files (of which some will be released a little later this week). For the first part of the exercise, '`chart.lsp`' provides a complete and functional implementation of the general chart parser (an improvement over the CKY parser), as discussed in the last two lectures.

- A this point, you should do virtually all your programming in the Lisp source files, i.e. '`chart.lsp`' for the first part of the exercise, and make sure to use emacs commands like 'M-C-x' or 'C-c C-b' to interactively load your code into the Lisp system.

2 Unpacking Parse Trees from Chart Edges

(a) The generalized chart parser computes a *parse forest* in polynomial time, i.e. a data structure that takes advantage of what we call 'packing' (or factoring) of local ambiguity: for each sub-string of input for which there are multiple ways of deriving the same category, the chart will only contain a single edge of that category. For example, when parsing the input *kim saw snow in oslo*, there are multiple ways of analyzing the VP *saw snow in oslo*. The specific context-free grammar we are assuming is provided at the top of '`chart.lsp`'; it closely resembles the simple grammar we have assumed in lectures. Inspect the grammar and, using pencil and paper, convince yourself that there multiple interpretations of this example. How many different parse trees are there for *kim saw snow in oslo*, given this grammar? Draw all valid trees and comment, in a sentence or two, on where exactly the ambiguities are located.

(b) Read through our file '`chart.lsp`' and its wealth of comments; make sure you understand the various abstract data types used, and look especially closely at the functions `parse()` and `fundamental-rule()`. Make sure you could explain the high-level structure of the parser to a friend, for example in an exam-like situation. Our Lisp code aligns quite closely with the more abstract presentation of the generalized chart parser used in our lecture slide copies. How exactly do we implement the notion of 'backpointers', i.e. the relation between an edge and its sequence of immediate daughters? Once you see the full picture, take a closer look at the function `pack-edge()` and understand what it means for one edge to be packed into a 'host' edge. Invoke the parser on an ambiguous input, for example:

```
? (parse '(kim saw snow in oslo))
```

Use the appropriate accessor function on the *edge* structure to retrieve the VP daughter from the parsing result and inspect that edge closely. Note that the global variable `*chart*` provides the complete parse chart once parsing has finished. Hence, another way of looking for the VP-level ambiguity in the parse chart would be to inspect the chart cell corresponding to the ambiguous sub-string. Use the function `chart-cell()` to find the same VP edge, i.e. the second daughter to the top-level `parse()` result for the current example.

(c) To *unpack* from the forest, loosely speaking, means to 'explode' an edge containing packed ambiguity into complete parse trees. In principle, there can be multiple packings deeply nested inside of daugther edges. For example an edge e with two daughers, x and y, may unpack into a total of six trees: if, in turn, x unpacks into three distinct trees, and y into two, then the sum of unpacked trees for e will be the cross-product of the two sets of daughter edges. To prepare for unpacking of edges, write a recursive

function `cross-product()` so that it takes as its single argument a list of lists (conceptually, a set of sets), and returns a set of all the tuples (again represented as lists), each of which contains exactly one element from each of the input sets; e.g.

```
? (cross-product '((1 2 3) (A B) (X Y)))
→ ((1 A X) (1 A Y) (1 B X) (1 B Y)
   (2 A X) (2 A Y) (2 B X) (2 B Y)
   (3 A X) (3 A Y) (3 B X) (3 B Y))
```

Because the number of sets to be multiplied out will depend on the number of daughters in each grammar rule, our definition of `cross-product()` must be able to process an arbitrary number of sets. What should be the base case, e.g. the return value for a call like:

```
? (cross-product '((1 2 3)))
```

Implement this function towards the end of the file '`chart.lsp`', where we have already provided the skeleton of the function, including a generous supply of helpful comments. Consider replacing the placeholder symbols '`???`' with actual code. There are many different ways of implementing this function, of course, so feel free to ignore our skeleton and write your function from scratch, if you find that easier. Be sure to test your function on sets of sets of varied sizes.

(d) Now look at the definition of `unpack-edge()` in '`chart.lsp`'. Given an edge, `unpack-edge()` returns a list of one or more trees corresponding to that edge once all 'packed' ambiguities have been multiplied out, for example:

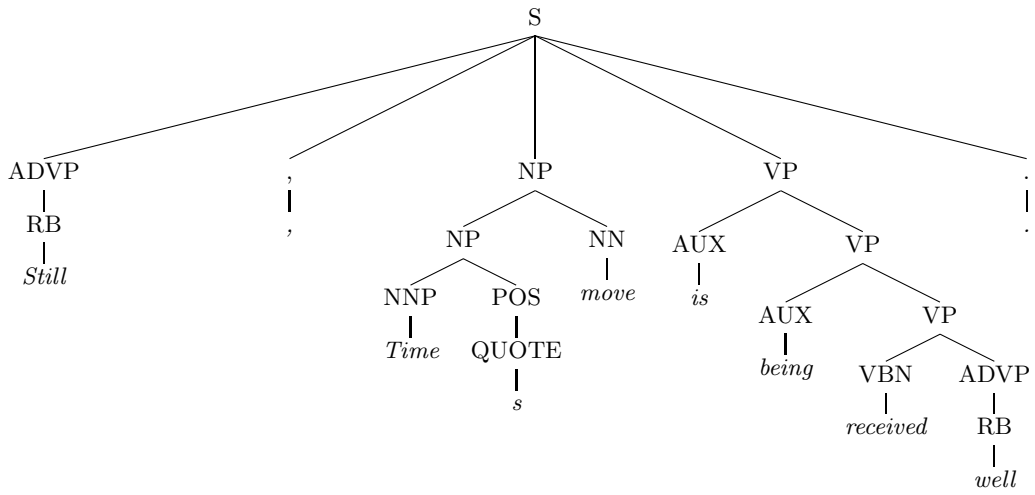
```
? (unpack-edge (first (parse '(kim saw snow in oslo))))
→ ((S (NP kim) (VP (VP (V saw) (NP snow)) (PP (P in) (NP oslo))))
   (S (NP kim) (VP (V saw) (NP (NP snow) (PP (P in) (NP oslo))))))
```

The implementation of `unpack-edge()` is a little tricky, since it has to look at both the *daughters* and *alternates* components of an *edge* structure. To write this function, you should have a good understanding of (a) how the *edge* structure works and (b) the techniques used for ambiguity packing in our chart parser. If in doubt, consult Chapter 14 of Jurafsky & Martin (2008) and the slides copies from the past two weeks. The general idea is straightforward, though: for edges without daughters, `unpack-edge()` returns a singleton list containing just the category of that edge (i.e. an 'atomic' tree or leaf node). For all other edges, `unpack-edge()` recurses over all daughters, cross multiplies all variations in each daughter position with all the other daughters, and builds as many trees as there are combinations; in addition to unfolding the current edge itself, `unpack-edge()` also needs to recurse over all equivalent edges (for which the current edge acts as a host) and combine the results of unpacking those with the list of trees corresponding to the host edge itself.

As for testing, again go with inputs of increasing complexity. You may choose to temporarily relax the selection of what is returned from the `parse()` function, so as to be able to parse just noun or verb phrases; if you were looking for an edge with a non-empty *alternates* value, consider parsing just *saw snow in oslo*—or use `chart-cell()` with appropriate indices after parsing a full sentence, as we did in part (b) above.

3 Theory: Probabilistic Context-Free Grammars

Consider the following parse tree for the sentence *Still, Time's move is being received well.* (taken from Section 23 of the Penn Treebank, PTB):



(a) Assuming this PTB tree was our complete training corpus, in one sentence sketch the procedure to extract context-free grammar (CFG) rules from the treebank. In another sentence, show how PCFG rule probabilities are estimated. When applying these procedures to the example above, which of the following rules are motivated by the treebank? Likewise, for those rules legitimately corresponding to our baby treebank, decide whether their probability estimates are correct (still assuming a training corpus comprised of only the above PTB tree), or determine the correct values where not.

CFG Rule	**PCFG Probability**
S → RB , NP NN AUX VP .	1.0
NP → NP NN	0.5
NP → NNP POS	0.5
VP → AUX VBN ADVP	0.5
VP → AUX AUX VBN ADVP	0.3
VP → AUX VP	0.5

4 Quasi-Destructive Graph Unification

For this part of the exercise, we provide three additional files: '`dag.lsp`', '`types.lsp`', and '`GLBS`'. We will make modifications to '`dag.lsp`'; the file '`types.lsp`' provides the definitions of a simple type hierarchy, organized as structures of type *type* (no pun intended). We already provide code in '`dag.lsp`' to read in the type hierarchy; the function `lookup-type()` can be used to retrieve an entry from the type hierarchy, taking the type name (a Lisp symbol) as its sole argument. We further provide a function `glb()` to compute the greatest lower bound of two types (specified by their names); for example:

```
? (glb 'noun-word '3sing-word)
→ NOUN-WORD-3SING
```

The definitions in '`types.lsp`' use structure components *parents* and *daughters* to spell out super- and sub-type relations. Find the entry for the type *pos*; recursively follow its sub-types and draw the complete type hierarchy below *pos*. Are there any instances of multiple inheritance in this sub-hierarchy? If so which types are involved.

The primary purpose of our type hierarchy, for the present problem set, is to provide the hierarchical relationships among feature structure types. However, each *type* entry in our hierarchy also has a component *dag*, which holds an example feature structure of that type. For testing purposes, we will use these feature structures as arguments to our unification and copy procedures.

(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`', consider the structure definition *dag*, with components *forward*, *type*, *arcs*, and *copy*, and also take a look at *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.

(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.

Note that we can look up the dags associated with entries in the type hierarchy and print them; for example:

```
? (pprint (type-dag (lookup-type 'head-initial)))
```

The feature structure associated with the type *head-initial* is especially interesting because it contains a so-called re-entrancy: the values of the two paths ⟨HEAD⟩ and ⟨ARGS FIRST HEAD⟩ are token-identical, i.e. both paths 'point' to the same *dag* object. Find the corresponding parts of the print-out produced by the above call: how does Lisp indicate re-entrancy of sub-structures?

To make visual inspection of feature structures—complex, recursive objects—a little easier, we provide a custom printer for *dag* objects: try using `print-dag()` instead of `pprint()` on our example dag. Compare the two outputs carefully.

(c) Go through the definition of `unify1()`, the main function of the unifier and fill in missing parts (incidated by '`???`' once again): `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; maybe experiment with the `glb()` interactively: call it with two symbols naming types from the hierary 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. Re-visit the *head-initial* example we inspected earlier; make sure to understand what it means for a feature structure traversal to 'visit' re-entrant nodes multiple times. Which of the nodes in this example will be visited more than once during unification, and how many times exactly?

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))))
```

What is the return value of this code fragment?

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 (a) the input dag is restored after the auxiliary function `copy1()` has been called and (b) that the dag copy is returned in the end.

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 internals of the type hierarchy 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 '`types.lsp`' file frequently, i.e. either re-evaluate the assignment of the global `*types*` variable or simply re-load the entire file '`dags.lsp`'.

(f) As indicated earlier, the value of the global variable `*types*` provides the hierarchy of types plus, for each type, a feature structure of that type (in the *dag* component of the *type* structure). 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))))
```

Remember to use `print-dag()` to obtain a complete, yet compact visual rendering of *dag* objects.

(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 your notes on the 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.

5 Submitting Your Results

Please submit your results in email to Stephan ('`oe@ifi.uio.no`') and Lars ('`larsbun@ifi.uio.no`') before 12:00 noon on Thursday, November 20. Please provide all files that you created as part of this exercise, including all code and answers to the questions above. Note that it is still good practice to generously document your code.