Algorithms for AI and NLP (Fall 2008)
— Sample Exam —

1 Finite-State Technology

(a) Draw the finite-state automaton (FSA) that corresponds to the regular expression `(a|b*)*`. Consider the language accepted by this FSA. How is it different from the automaton `(a|b)*`, if at all?

(b) For each of the following strings, say whether it is part of the language recognized by this automation or not: (i) `aaa`, (ii) `bab`, (iii) `abab`, (iv) `aabaab`.

(c) Recall the distinction between deterministic and non-deterministic FSAs. What does it mean for an FSA to be non-deterministic? Is your solution to part (a) above deterministic or not? In a few sentences, sketch the procedure for converting a non-deterministic FSA into a deterministic one. In the general case, what is the maximum number of states in the new, deterministic automaton, assuming the original non-deterministic FSA had n states?

2 Language Modelling

(a) How exactly does an n-gram language model estimate the probability $P(s)$ for a string $s = w_1^n$, assuming a first-order n-gram model. Discuss the central assumption made in this modelling approach; what is the common name of said assumption?

(b) What is the formula for estimating n-gram probabilities from a training corpus of running text; show the calculations for uni- and bi-grams.

(c) In a few sentences, discuss the problem of data sparseness and its consequences for a naïve application of an n-gram language model. What is the general idea common to various smoothing schemes. Explain the particular method of add-one (or Laplace) smoothing.

3 Context-Free Grammars and Parsing

Consider the language defined by the following grammar (assuming, by convention, 'S' as the start symbol):

S → NP VP	NP → *kim*
VP → VP PP	NP → *oslo*
NP → NP PP	NP → *snow*
VP → V NP	V → *adores*
VP → V	V → *snores*
PP → P NP	P → *in*

(a) For each of the following sentences, identify the number of readings (distinct analyses) that this grammar will assign, and draw the parse trees for each of the readings.

(i) *kim in oslo adores snow in oslo*

(ii) *kim snores in oslo*

(iii) *kim snores snow in oslo*

(b) Does this grammar contain rules that would be problematic for our naïve, top-down parser? If so, identify the(se) rule(s) and what exactly would constitute a problem for the top-down parser? Discuss the properties of this grammar that either support or prevent the use of a CKY parser

(c) Which aspect(s) of natural language make(s) parsing (using context-free grammars, say) a surprisingly hard problem? How is a naïve parser affected in its performance, and (quite generally speaking) what is the magnitude of its worst-case complexity relative to input string length? Mention at least one example.

(d) How does our generalized chart parser avoid this worst-case complexity, in general and for the example you gave above? Sketch, in a few sentences, the central idea in chart-based parsing.

4 Feature Structures and Unifiction

(a) Draw the following feature structure in DAG notation, i.e. as a graph of labeled nodes and directed arcs:

$$\textit{phrase}\left[\begin{array}{ll}\texttt{HEAD}\ \boxed{1}\ \textit{verb} & \\ \texttt{ARGS} & \textit{*ne-list*}\left[\begin{array}{ll}\texttt{FIRST} & \textit{word}\left[\begin{array}{l}\texttt{ORTH}\ \textit{“chased”}\\ \texttt{HEAD}\ \boxed{1}\end{array}\right]\\ \texttt{REST} & \textit{*ne-list*}\left[\begin{array}{ll}\texttt{FIRST} & \textit{sign}\left[\texttt{HEAD}\ \textit{noun}\right]\\ \texttt{REST} & \textit{*null*}\end{array}\right]\end{array}\right]\end{array}\right]$$

(b) Does this feature structure contain what we called co-references or re-entrancies? If so, for the DAG representation of the structure, what exactly does it mean for a graph to be re-entrant?

(c) In a sentence or two, sketch the challenge in creating a structually equivalent copy of a re-entrant DAG. How does our copy algorithm solve this problem?

(d) In a few sentences, recall the notions of *early-* and *over-copying*. In your discussion, contrast destructive vs. non-destructive unification algorithms, and also consider the case of unification failures.

(e) Sketch the basic idea of quasi-destructive dag manipulation, say when applied to the copy operation. Does a generation counting scheme applied to the *copy* slot of our *dag* structures (and only to that slot) reduce either of early- or over-copying? If so, how? If not, why not?

5 Common-Lisp

(a) Describe the effects of the function `?()` below. Discuss at least one calling example and show the ‘box notation’ used to keep track of `cons()` cells.

```
(defun ? (foo)
  (if (atom foo)
    foo
    (cons (? (first foo)) (? (rest foo)))))
```

(b) Write a two-place function `ditch()` that takes an atom as its first and a list as its second argument; `ditch()` removes *all* occurrences of the atom in the list, e.g.

```
? (ditch 'c '(a b c d e c))
→ (A B D E)
? (ditch 'f '(a b c d e c))
→ (A B C D E C)
```

Note that we are not primarily concerned with specific details of Lisp syntax here. If you found that easier, feel free to use elements of ‘pseudo code’ in your function definition, as long as it is clear how exactly everything will work.

(c) Sketch a revision of `ditch()` that does not generate new `cons()` cells, i.e. avoids all calls to functions like `cons()`, `append()`, or `list()`. In proposing your revised implementation, give a brief informal summary of the basic principles (e.g. in terms of base case(s) vs. recursive cases) and general approach.