UNIVERSITY OF CALIFORNIA
Department of Electrical Engineering
and Computer Sciences
Computer Science Division

CS 294-8 **Prof. R. Fateman**
Fall 2006

Assignment 3: Pattern Matching, Search

Due: Oct 2, 2006

1. Read chapters 5,6,7 in Norvig. If you somehow picked up an old (1st printing) of the text, check out the Errata in the on-line files.

2. How would you change Eliza, especially in view of the more elaborate version of Eliza that could be easily constructed using our pattern matcher, so that it can respond to simple arithmetic queries such as "How much is 2 + 2" (try this with google!) or simple computer-related questions such as "What is my current working directory"? (assume that there is a variant of the lisp command `(run-shell-command "pwd")` that makes it possible to find out such facts. Describe in enough detail so that it is clear you could program this.

Historically, the informed view seemed to be that this is not a promising approach to natural language understanding, because this program does not seem to have a model for what is going on. Yet with enough mechanism to choose "which rule to fire" this has a certain resemblance to some "expert systems" that enjoyed a burst of popularity in the 1980s. Discuss how one might get more "expert" behavior out of Eliza.

3. Re-write the interactive interpreter on Norvig's page 178 (and 216) using a different technique based on `with-simple-restart`. If you look at Steele's *Common Lisp the Language* (2nd ed.) page 902, you'll see how.

4. (Commutative matching) As discussed in class, in some pattern-matching applications it is important to be able to deal with commutative operators like "OR", "AND", "+", and "*" For example you might wish to write a single rule that simplifies `(OR ?x (AND A ?x))` to just `?x`, logically. This rule should apply to `(OR B (AND B A)`. Explain how one could do this with specific reference to the existing programs in Norvig's matcher, or the variation of the matcher shown below. How does this affect the speed of matches?

5. (Missing defaults) A pattern `(* ?x (+ ?y ?z))` could match `(+ b c)` by matching `?x` against an (implicit) 1. That is, `(+ b c)` could be treated as though it were `(* 1 (+ b c))`. Explain at least one way to do this. How does it affect the speed of matches?

6. Exercise 6.10 page 215 in Norvig.

7. Provide one or perhaps several proposals for your class project. A half-page description of your goals and approach should be enough at this time. If you need help, I'm available to discuss the project list, clarify the scope of a project that is appropriate for the class, etc.

```
(defun pat-match (pattern input stk-ptr cfn)
  ;; uses continuation function and separate operator/operand matching
  ;; uses a stack instead of bindings -- stk-ptr could be just an
  ;; integer. In this program section, the details are irrelevant.
  "Match pattern against input in the context of the stk-ptr"
  (cond ((eq stk-ptr fail) fail)
        ((variable-p pattern)
         (match-variable pattern input stk-ptr cfn))
        ((eql pattern input) (funcall cfn stk-ptr)) ;;** note this
        ((segment-pattern-p pattern)
         (segment-matcher pattern input stk-ptr cfn))
        ((single-pattern-p pattern)
         (single-matcher pattern input stk-ptr cfn))
        ;; in the earlier version, we just recursed down the head and rest
        ;; of the pattern and input.  This no longer works if we assume
        ;; the pattern and input have the structure:
        ;; (operator operand_1 operand_2 ... operand_n)
        ;; because the matching of the list of operands (operand_1 ...)
        ;; may be affected by the governing operator. We insert this line:

        ((and (consp pattern)(special-pattern-p (car pattern)))
         (funcall (special-pattern-p (car pattern))
                  pattern input stk-ptr cfn))
        ;; now we try to show how to back up past successful matches
        ;; that fail as a result of later requirements.
        ;;we have 3 obligations: match 1st to 1st, match rest to rest
        ;;and make sure the continuation function cfn returns success.

        ((and (consp pattern) (consp input)
         (setq stk-ptr
            (pat-match (first pattern)(first input) stk-ptr
                              #'(lambda(s)
                                  (and (pat-match (rest pattern)
                                                  (rest input)
                                                  s
                                                  cfn))))))

         stk-ptr)
        (t fail)))
```