```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;PARSE a subset of English sentences
;;;
;;;  code written by an accumulation of teaching assistants and
;;;   students in Stanford's CS102 LISP Programming classes 1980-83.

(defvar *nouns*
  '(birds john mary dog cat mice bird hand bush man sea fish ketchup
    I girl tree hill telescope store ideas hats))
(defvar *determiners*
  '(a an the your))
(defvar *verbs*
  '(fly loves chased worth eating saw sleep))
(defvar *prepositions*
  '(in by with on from))
(defvar *verb-auxiliaries*
  '(is are was were be been being has have had do does did will shall could))
(defvar *adjectives*
  '(big black seven frightened old raw hungry furious green))
(defvar *lexicon*
  (append *nouns* *determiners* *verbs* *prepositions*
          *verb-auxiliaries* *adjectives*))


(defun parse (s)
  (let ((lex-fail (check-lexicon s)))
    (if lex-fail
      (format nil "These words are not in the lexicon:~%   ~S" lex-fail)
      (outnil (parse-sentence s)))))

(defun parse-sentence (s)
  ((lambda (s-pair)
     (cond ((and s-pair (null (rest-part s-pair)))
            (parse-part s-pair))
           (T '*UNGRAMMATICAL*)))
   (sentence s)))

;;; <S>   -->  <NP> <VP>
(defun sentence (s)
  ((lambda (np-pair)
     (cond (np-pair
            ((lambda (vp-pair)
               (cond (vp-pair
                      (cons (list '*SENT* '*SUBJ* (parse-part np-pair)
                                  '*PRED* (parse-part vp-pair))
                            (rest-part vp-pair)))
                     (T nil)))
             (verb-phrase (rest-part np-pair))))
           (T nil)))
   (noun-phrase s)))

;;; <NP>  -->  [<detr>] <adjv>* <noun> <PP>*
(defun noun-phrase (np)
```

```
  ((lambda (detr-pair)
     ((lambda (adjv-pair)
        ((lambda (noun-pair)
           (cond (noun-pair
                  ((lambda (pp-pair)
                     (cons (list '*NP* (parse-part noun-pair)
                                 '*DETR* (parse-part detr-pair)
                                 '*ADJV* (parse-part adjv-pair)
                                 '*MODS* (parse-part pp-pair))
                           (rest-part pp-pair)))
                   (find-* #'prep-phrase (rest-part noun-pair))))
                 (T nil)))
         (noun (rest-part adjv-pair))))          ;gives noun-pair
      (find-* #'adjv (rest-part detr-pair))))    ;gives adjv-pair
   (find-? #'detr np)))                          ;gives detr-pair

;;; <VP>  -->  <vaux>* <verb> [<NP>] <PP>*
(defun verb-phrase (vp)
  ((lambda (vaux-pair)
     ((lambda (verb-pair)
        (cond (verb-pair
               ((lambda (np-pair)
                        ((lambda (pp-pair)
                           (cons (list '*VP* (parse-part verb-pair)
                                       '*AUX* (parse-part vaux-pair)
                                       '*OBJ* (parse-part np-pair)
                                       '*MODS* (parse-part pp-pair))
                                 (rest-part pp-pair)))
                         (find-* #'prep-phrase (rest-part np-pair))))
                (find-? #'noun-phrase (rest-part verb-pair))))
              (T nil)))
      (verb (rest-part vaux-pair))))
   (find-* #'vaux vp)))

;;; <PP>  -->  <prep> <NP>
(defun prep-phrase (pp)
  ((lambda (prep-pair)
     (cond (prep-pair
            ((lambda (np-pair)
               (cond (np-pair
                      (cons (list '*PP* (parse-part prep-pair)
                                  '*OBJ* (parse-part np-pair))
                            (rest-part np-pair)))
                     (t nil)))
             (noun-phrase (rest-part prep-pair))))
           (t nil)))
   (prep pp)))


(defun noun (w) (get-word 'noun w))
(defun detr (w) (get-word 'detr w))
(defun verb (w) (get-word 'verb w))
(defun prep (w) (get-word 'prep w))
(defun vaux (w) (get-word 'vaux w))
```

```
(defun adjv (w) (get-word 'adjv w))


;;; FIND-* returns a list of parses of TYPE and what is left
(defun find-* (type frag)
  ((lambda (type-pair)
     (cond (type-pair
            ((lambda (find-rest)
               (cons (cons (parse-part type-pair)
                           (parse-part find-rest))
                     (rest-part find-rest)))
             (find-* type (rest-part type-pair))))
           (T (cons nil frag))))
   (funcall type frag)))

;;; FIND-? returns (parse . rest) when TYPE is matched
(defun find-? (type frag)
  (cond ((funcall type frag))
        (T (cons nil frag))))

;;; GET-WORD finds a word of a given type,
;;;   returning a pair of the word and the rest of the fragment
(defun get-word (type frag)
  (cond ((null frag) nil)
        ((isa (first frag) type) frag)
        (T nil)))

;;; REST-PART returns the rest of the fragment
;;;   after the parsed part is removed
(defun rest-part (pair)
  (rest pair))

;;; PARSE-PART returns the parsed part
(defun parse-part (pair)
  (first pair))

(defun isa (word type)
  (member word
          (case type
            (noun *nouns*)
            (detr *determiners*)
            (verb *verbs*)
            (prep *prepositions*)
            (vaux *verb-auxiliaries*)
            (adjv *adjectives*)
            (otherwise nil)
            )))

;;;finally here is a post-processing function which cleans up
;;;  the output to delete categories which are empty (ie = nil)

(defun outnil (x)
  (cond ((atom x) x)
        ((and (rest x) (atom (first x)) (null (second x)))
```

```
         (outnil (rest (rest x))))
        (T (cons (outnil (first x))
                 (outnil (rest x))))))

;;;and a pre-processing function which checks that the words in
;;;  a sentence are in the lexicon.

(defun check-lexicon (s)
  (let ((fails nil))
    (dolist (i s fails)
      (when (not (member i *lexicon*))
        (push i fails)))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;ADDITIONAL ABSTRACTION
;;;  PARSE-FIND takes a list of occurence types,
;;;    (<...>, <...>*, [...])
;;;    and manages the entire parsing operation defined by the
;;;    metalanguage of parsing rules
;;;
;;; the parsing categories also need to be written in list form
;;;   <...>*  =  (* ...)
;;;   [...]   =  (? ...)

(defvar *rest* nil)
(defun parse-find (categories s)
  (cond ((null categories)
         (cons s nil))
        (((lambda (pair)
            (cond ((and pair
                        (setq *rest* (find (rest categories)
                                           (rest-part pair))))
                   (cons (cons (cons (first categories)
                                     (parse-part pair))
                               (parse-part *rest*))
                         (rest-part *rest*)))
                  (T nil)))
          (find-category (first categories) s)))
        (T nil)))

(defun find-category (cat s)
  (cond ((eq (prefix cat) '?)
         (find-? (suffix cat) s))
        ((eq (prefix cat) '*)
         (find-* (suffix cat) s))
        (T (funcall cat s))))

(defun prefix (type)
  (first type))

(defun suffix (type)
  (second type))
```

```
#|
;;; to use the new PARSE-FIND, each specialized parsing function
;;;   needs to be rewritten in its simpler, more abstract form
;;; For instance, the parsing function SENTENCE would now be

;;; <S>   -->  <NP> <VP>
(defun abstract-sentence (s)
  (let ((category (parse-find '(np vp) s)))
    (cond (category
           (cons (list '*S* '*SUBJ* (get-parse 'NP category)
                       '*PRED* (get-parse 'VP category))
                 (rest-part category)))
          (T nil))))

(defun get-parse (type cat)
  (cdr (assq type (car cat))))
|#


#|
;;; try these test sentences

(birds fly)  ==>
(*SENT* *SUBJ* (*NP* birds)
        *PRED* (*VP* fly))

(john loves mary)
==>
(*SENT* *SUBJ* (*NP* john)
        *PRED* (*VP* loves *OBJ* (*NP* mary)))

(a dog chased the cat)
==>
(*SENT* *SUBJ* (*NP* dog *DETR* a)
        *PRED* (*VP* chased *OBJ* (*NP* cat *DETR* the)))

(the big black dog chased seven frightened mice)
==>
(*SENT* *SUBJ* (*NP* dog *DETR* the *ADJV* (big black))
        *PRED* (*VP* chased
                     *OBJ* (*NP* mice *ADJV* (seven frightened))))

(a bird in hand is worth two in the bush)
==>
*UNGRAMMATICAL*

(a bird in hand is worth two birds in the bush)
==>
(*SENT* *SUBJ* (*NP* bird *DETR* a *MODS* ((*PP* in *OBJ* (*NP* hand))))
        *PRED* (*VP* worth *AUX* (is)
                     *OBJ* (*NP* birds *ADJV* (two)
    *MODS* ((*PP* in *OBJ* (*NP* bush *DETR* the))))))

(the old man by the sea could have been eating raw fish with ketchup)
```

```
==>
(*SENT* *SUBJ* (*NP* MAN *DETR* THE *ADJV* (OLD)
                     *MODS* ((*PP* BY *OBJ* (*NP* SEA *DETR* THE))))
        *PRED* (*VP* EATING *AUX* (COULD HAVE BEEN)
                     *OBJ* (*NP* FISH *ADJV* (RAW)
                                 *MODS* ((*PP* WITH *OBJ* (*NP* KETCHUP))))))

(I saw the girl by the big tree on the hill with a telescope from your store)
==>

(*SENT* *SUBJ* (*NP* I)
        *PRED* (*VP* saw
                     *OBJ* (*NP* girl *DETR* the
    *MODS* ((*PP* by *OBJ* (*NP* tree *DETR* the *ADJV* (big)
      *MODS* ((*PP* on *OBJ* (*NP* hill *DETR* the
        *MODS* ((*PP* with *OBJ* (*NP* telescope *DETR* a
          *MODS* ((*PP* from *OBJ* (*NP* store *DETR* your))))))))))))))

;;;a non-parsible sentence.  We have not allowed for adverbs (or for
;;; pronouns), so have no category for "quickly" in the lexicon.
(Birds fly quickly)

;;;a parsible non-sentence
(The I be be be loves)
==>
(*SENT* *SUBJ* (*NP* I *DETR* THE)
        *PRED* (*VP* LOVES *AUX* (BE BE BE)))

;;;a parsible nonsence sentence
(Furious green ideas sleep with hungry hats)
==>
(*SENT* *SUBJ* (*NP* IDEAS *ADJV* (FURIOUS GREEN))
        *PRED* (*VP* SLEEP
    *MODS* ((*PP* WITH *OBJ* (*NP* HATS *ADJV* (HUNGRY))))))

|#
```