```
;;;================================
;;; -*- Mode: Lisp; Syntax: Common-Lisp; -*-
;;; Code from Paradigms of Artificial Intelligence Programming
;;; Copyright (c) 1991 Peter Norvig


(defun variable-p (x)
  "Is x a variable (a symbol beginning with `?')?"
  (and (symbolp x) (equal (char (symbol-name x) 0) #\?)))

(defconstant fail nil "Indicates pat-match failure")

(defconstant no-bindings '((t . t))
  "Indicates pat-match success, with no variables.")

;;; ==============================

(defun get-binding (var bindings)
  "Find a (variable . value) pair in a binding list."
  (assoc var bindings))

(defun binding-val (binding)
  "Get the value part of a single binding."
  (cdr binding))

(defun lookup (var bindings)
  "Get the value part (for var) from a binding list."
  (binding-val (get-binding var bindings)))

(defun extend-bindings (var val bindings)
  "Add a (var . value) pair to a binding list."
  (cons (cons var val)
        ;; Once we add a "real" binding,
        ;; we can get rid of the dummy no-bindings
        (if (and (eq bindings no-bindings))
            nil
            bindings)))

;;; ==============================

(defun pat-match (pattern input &optional (bindings no-bindings))
  "Match pattern against input in the context of the bindings"
  (cond ((eq bindings fail) fail)
        ((variable-p pattern)
         (match-variable pattern input bindings))
        ((eql pattern input) bindings)
        ((segment-pattern-p pattern)                ; ***
         (segment-match pattern input bindings))    ; ***
        ((and (consp pattern) (consp input))
         (pat-match (rest pattern) (rest input)
                    (pat-match (first pattern) (first input)
                               bindings)))
```

```
        (t fail)))

(defun match-variable (var input bindings)
  "Does VAR match input?  Uses (or updates) and returns bindings."
  (let ((binding (get-binding var bindings)))
    (cond ((not binding) (extend-bindings var input bindings))
          ((equal input (binding-val binding)) bindings)
          (t fail))))

(defun segment-pattern-p (pattern)
  "Is this a segment matching pattern: ((?* var) . pat)"
  (and (consp pattern)
       (starts-with (first pattern) '?*)))

;;; ==============================

(defun segment-match (pattern input bindings &optional (start 0))
  "Match the segment pattern ((?* var) . pat) against input."
  (let ((var (second (first pattern)))
        (pat (rest pattern)))
    (if (null pat)
        (match-variable var input bindings)
        ;; We assume that pat starts with a constant
        ;; In other words, a pattern can't have 2 consecutive vars
        (let ((pos (position (first pat) input
                             :start start :test #'equal)))
          (if (null pos)
              fail
              (let ((b2 (pat-match
                         pat (subseq input pos)
                         (match-variable var (subseq input 0 pos)
                                         bindings))))
                ;; If this match failed, try another longer one
                (if (eq b2 fail)
                    (segment-match pattern input bindings (+ pos 1))
                    b2)))))))

;;; ==============================

(defun rule-pattern (rule) (first rule))
(defun rule-responses (rule) (rest rule))

;;; ==============================

(defparameter *eliza-rules*
 '((((?* ?x) hello (?* ?y))
    (How do you do.  Please state your problem.))
   (((?* ?x) computer (?* ?y))
    (Do computers worry you?) (What do you think about machines?)
    (Why do you mention computers?)
    (What do you think machines have to do with your problem?))
   (((?* ?x) name (?* ?y))
    (I am not interested in names))
   (((?* ?x) sorry (?* ?y))
```

```
 (Please don't apologize) (Apologies are not necessary)
 (What feelings do you have when you apologize))
(((?* ?x) I remember (?* ?y))
 (Do you often think of ?y)
 (Does thinking of ?y bring anything else to mind?)
 (What else do you remember) (Why do you recall ?y right now?)
 (What in the present situation reminds you of ?y)
 (What is the connection between me and ?y))
(((?* ?x) do you remember (?* ?y))
 (Did you think I would forget ?y ?)
 (Why do you think I should recall ?y now)
 (What about ?y) (You mentioned ?y))
(((?* ?x) if (?* ?y))
 (Do you really think its likely that ?y) (Do you wish that ?y)
 (What do you think about ?y) (Really-- if ?y))

(((?* ?x) I dreamt (?* ?y))
 (Really-- ?y) (Have you ever fantasized ?y while you were awake?)
 (Have you dreamt ?y before?))
(((?* ?x) dream about (?* ?y))
 (How do you feel about ?y in reality?))
(((?* ?x) dream (?* ?y))
 (What does this dream suggest to you?) (Do you dream often?)
 (What persons appear in your dreams?)
 (Don't you believe that dream has to do with your problem?))
(((?* ?x) my mother (?* ?y))
 (Who else in your family ?y) (Tell me more about your family))
(((?* ?x) my father (?* ?y))
 (Your father) (Does he influence you strongly?)
 (What else comes to mind when you think of your father?))

(((?* ?x) I want (?* ?y))
 (What would it mean if you got ?y)
 (Why do you want ?y) (Suppose you got ?y soon))
(((?* ?x) I am glad (?* ?y))
 (How have I helped you to be ?y) (What makes you happy just now)
 (Can you explain why you are suddenly ?y))
(((?* ?x) I am sad (?* ?y))
 (I am sorry to hear you are depressed)
 (I'm sure its not pleasant to be sad))
(((?* ?x) are like (?* ?y))
 (What resemblance do you see between ?x and ?y))
(((?* ?x) is like (?* ?y))
 (In what way is it that ?x is like ?y)
 (What resemblance do you see?)
 (Could there really be some connection?) (How?))
(((?* ?x) alike (?* ?y))
 (In what way?) (What similarities are there?))
(((?* ?x) same (?* ?y))
 (What other connections do you see?))

(((?* ?x) I was (?* ?y))
 (Were you really?) (Perhaps I already knew you were ?y)
 (Why do you tell me you were ?y now?))
```

```
(((?* ?x) was I (?* ?y))
 (What if you were ?y ?) (Do you thin you were ?y)
 (What would it mean if you were ?y))
(((?* ?x) I am (?* ?y))
 (In what way are you ?y) (Do you want to be ?y ?))
(((?* ?x) am I (?* ?y))
 (Do you believe you are ?y) (Would you want to be ?y)
 (You wish I would tell you you are ?y)
 (What would it mean if you were ?y))
(((?* ?x) am (?* ?y))
 (Why do you say "AM?") (I don't understand that))
(((?* ?x) are you (?* ?y))
 (Why are you interested in whether I am ?y or not?)
 (Would you prefer if I weren't ?y)
 (Perhaps I am ?y in your fantasies))
(((?* ?x) you are (?* ?y))
 (What makes you think I am ?y ?))

(((?* ?x) because (?* ?y))
 (Is that the real reason?) (What other reasons might there be?)
 (Does that reason seem to explain anything else?))
(((?* ?x) were you (?* ?y))
 (Perhaps I was ?y) (What do you think?) (What if I had been ?y))
(((?* ?x) I can't (?* ?y))
 (Maybe you could ?y now) (What if you could ?y ?))
(((?* ?x) I feel (?* ?y))
 (Do you often feel ?y ?))
(((?* ?x) I felt (?* ?y))
 (What other feelings do you have?))
(((?* ?x) I (?* ?y) you (?* ?z))
 (Perhaps in your fantasy we ?y each other))
(((?* ?x) why don't you (?* ?y))
 (Should you ?y yourself?)
 (Do you believe I don't ?y) (Perhaps I will ?y in good time))
(((?* ?x) yes (?* ?y))
 (You seem quite positive) (You are sure) (I understand))
(((?* ?x) no (?* ?y))
 (Why not?) (You are being a bit negative)
 (Are you saying "NO" just to be negative?))

(((?* ?x) someone (?* ?y))
 (Can you be more specific?))
(((?* ?x) everyone (?* ?y))
 (surely not everyone) (Can you think of anyone in particular?)
 (Who for example?) (You are thinking of a special person))
(((?* ?x) always (?* ?y))
 (Can you think of a specific example) (When?)
 (What incident are you thinking of?) (Really-- always))
(((?* ?x) what (?* ?y))
 (Why do you ask?) (Does that question interest you?)
 (What is it you really want to know?) (What do you think?)
 (What comes to your mind when you ask that?))
(((?* ?x) perhaps (?* ?y))
 (You do not seem quite certain))
```

```
   (((?* ?x) are (?* ?y))
    (Did you think they might not be ?y)
    (Possibly they are ?y))
   (((?* ?x))
    (Very interesting) (I am not sure I understand you fully)
    (What does that suggest to you?) (Please continue) (Go on)
    (Do you feel strongly about discussing such things?))))

;;; ==============================

(defun eliza ()
  "Respond to user input using pattern matching rules."
  (loop
    (print 'eliza>)
    (let* ((input (read-line-no-punct))
           (response (flatten (use-eliza-rules input))))
      (print-with-spaces response)
      (if (equal response '(good bye)) (RETURN)))))

(defun use-eliza-rules (input)
  "Find some rule with which to transform the input."
  (some #'(lambda (rule)
            (let ((result (pat-match (rule-pattern rule) input)))
              (if (not (eq result fail))
                  (sublis (switch-viewpoint result)
                          (random-elt (rule-responses rule))))))
        *eliza-rules*))

(defun switch-viewpoint (words)
  "Change I to you and vice versa, and so on."
  (sublis '((I . you) (you . I) (me . you) (am . are))
          words))

;;; ==============================

(defun read-line-no-punct ()
  "Read an input line, ignoring punctuation."
  (read-from-string
    (concatenate 'string "(" (substitute-if #\space #'punctuation-p
                                            (read-line))
                 ")")))

(defun punctuation-p (char) (find char ".,;:`!?#-()\\\""))

(defun print-with-spaces (list)
  (mapc #'(lambda (x) (prin1 x) (princ " ")) list))

(defun print-with-spaces (list)
  (format t "~{~a ~}" list))

;;; ==============================


(defun flatten (the-list)
```

```
  "Append together elements (or lists) in the list."
  (mappend #'mklist the-list))

(defun mklist (x)
  "Return x if it is a list, otherwise (x)."
  (if (listp x)
      x
      (list x)))

(defun mappend (fn the-list)
  "Apply fn to each element of list and append the results."
  (apply #'append (mapcar fn the-list)))

(defun random-elt (choices)
  "Choose an element from a list at random."
  (elt choices (random (length choices))))

(defun starts-with (list x)
  "Is x a list whose first element is x?"
  (and (consp list) (eql (first list) x)))

;;; ==============================

;;;? (eliza)
;;;ELIZA> hello
;;;HOW DO YOU DO. PLEASE STATE YOUR PROBLEM.
;;;ELIZA> my mother is hungry
;;;WHO ELSE IN YOUR FAMILY IS HUNGRY
;;;ELIZA> I am
;;;IN WHAT WAY ARE YOU
;;;ELIZA> very
;;;GO ON
;;;ELIZA> Gee i want something to eat
;;;WHY DO YOU WANT SOMETHING TO EAT
;;;ELIZA> I'm hungry
;;;DO YOU FEEL STRONGLY ABOUT DISCUSSING SUCH THINGS?
;;;ELIZA> yes, I want some candy
;;;WHAT WOULD IT MEAN IF YOU GOT SOME CANDY
;;;ELIZA> I'd be happy
;;;VERY INTERESTING
;;;ELIZA> quit
;;;GO ON
;;;ELIZA> exit
;;;I AM NOT SURE I UNDERSTAND YOU FULLY
;;;ELIZA> stop
;;;GO ON
;;;ELIZA> [Aborted]

;;; ==============================
```

```
;;; -*- Package: User; Syntax: Common-Lisp; Mode: Lisp; Base: 10 -*-
;;; unify.lisp

;;; ************************************************************
;;; Unification Algorithm ****************************************
;;; ************************************************************
;;;
;;; Random implementations of unification.
;;;
;;; Written by Mark Kantrowitz, mkant@cs.cmu.edu, October 15, 1990.
;;;
;;; Sorry, no documentation.
;;;
;;; To do:
;;;    lazy unification

;;; *******************************
;;; Global Variables ***************
;;; *******************************
(defvar *failure* 'failed)

;;; *******************************
;;; Macros *************************
;;; *******************************
(defmacro xor (a b)
  `(or (and ,a (not ,b))
       (and ,b (not ,a))))

(defmacro nor (a b)
  `(and (not ,a) (not ,b)))

;;; *******************************
;;; Primitives *********************
;;; *******************************
(defun occurs (elt lst)
  "Returns t if elt occurs somewhere in lst"
  (cond ((null lst)
         nil)
        ((listp lst)
         (or (occurs elt (car lst))
             (occurs elt (cdr lst))))
        ((atom lst)
         (eq lst elt))))

;;; *******************************
;;; Variables **********************
;;; *******************************
(defun make-variable (x)
  (make-symbol (format nil "?~a" x)))

(defun variablep (item)
  "A variable is of the form ?name, where name is a symbol."
```

```
  (and (symbolp item)
       (char= (char (symbol-name item) 0)
              #\?)))

(defun variable-lookup (var env)
  (let* ((binding (assoc var env))
         (val (cdr binding)))
    (cond ((variablep val)
           (variable-lookup val env))
          ((null binding)
           ;; Unbound variable, so the variable itself is returned.
           var)
          ((null val)
           ;; Null variable value.
           nil)
          ((and (not (occurs var val))
                (apply-substitutions env val)))
          (t val))))

(defun apply-substitutions (substitutions elt)
  (cond ((null elt)
         nil)
        ((listp elt)
         (cons (apply-substitutions substitutions (car elt))
               (apply-substitutions substitutions (cdr elt))))
        ((variablep elt)
         (variable-lookup elt substitutions))
        (t elt)))


;;; ******************************
;;; Recursive Unification **********
;;; ******************************
;;; See Rich & Knight, p. 181

(defun recursive-unify (l1 l2)
  (cond ((or (atom l1) (atom l2))
         (cond ((eq l1 l2)
                nil)
               ((variablep l1)
                (if (occurs l1 l2)
                    *failure*
                    (list (cons l1 l2))))
               ((variablep l2)
                (if (occurs l2 l1)
                    *failure*
                    (list (cons l2 l1))))
               (t
                *failure*)))
        ((not (= (length l1) (length l2)))
         *failure*)
        (t
         (let ((subst nil))
           (do* ((sl1 l1 (cdr sl1))
```

```
                (e1 (car sl1) (car sl1))
                (sl2 l2 (cdr sl2))
                (e2 (car sl2) (car sl2)))
               ((null sl1)
                subst)
             (let ((s (recursive-unify e1 e2)))
               (cond ((eq s *failure*)
                      (return *failure*))
                     (s
                      (setf sl1 (apply-substitutions s sl1))
                      (setf sl2 (apply-substitutions s sl2))
                      (setq subst (append s subst)))))))))


;;; *******************************
;;; Iterative Unfication ***********
;;; *******************************

(defun unify (pattern data &optional env trace)
  "This is a fast iterative implementation of unification. It eliminates
   the need for a stack in a manner similar to tail-recursion. We model the
   flow of control in unification by saving untested pattern and data elements
   on a \"continuation stack\". At any point of the program, we are either
   updating the iteration variables or testing a pattern element against
   a data element (which must then be either atoms or variables). If this
   test fails, we return *failure* immediately. Otherwise, we accumulate
   any substitutions in the environment, which will ultimately be returned."
  (let ((rest-pattern nil)              ; these act as continuations
        (rest-data nil)
        binding)
    (loop
     (when trace
       ;; For debugging.
       (format t "~&Pattern:~T~A  ~&Data:~T~A  ~&Environment:~T~A"
               pattern data env))
     (cond ((or (and pattern (atom pattern))
                (and data (atom data)))
            ;; We have a pattern and a data to match, at least one
            ;; of which is a non-nil atom.
            (cond ((eq pattern data)
                   ;; If pattern and data are identical, test next elements.
                   (setf data nil pattern nil))
                  ;; Note: we aren't doing any sort of occurrence check
                  ;; to see if variable lookup will lead to infinite
                  ;; loops. For example, (?a ?b) against (?b ?a), or
                  ;; even ?a against (b ?a).
                  ((variablep data)
                   ;; Lookup the variable, if possible.
                   (setf binding (assoc data env))
                   (if binding
                       ;; If there's a data binding, substitute and try again.
                       (setf data (cdr binding))
                     ;; If no data binding, add one and move on.
                     (setf env (acons data pattern env)
```

```
                        data nil pattern nil)))
                ((variablep pattern)
                 (setf binding (assoc pattern env))
                 (if binding
                     (setf pattern (cdr binding))
                   (setf env (acons pattern data env)
                         data nil pattern nil)))
                (t
                 ;; Match failed. Probably because of data-pattern mismatch.
                 (return *failure*))))
          ((nor pattern data)
           ;; If we've run out of pattern and data (both nil), check the
           ;; rest-pattern and rest-data.
           (cond ((xor rest-pattern rest-data)
                  ;; If we have a mismatch, fail.
                  (return  *failure*))
                 ((nor rest-pattern rest-data)
                  ;; If we've run out there too, exit with the bindings.
                  (return env))
                 (t
                  ;; Otherwise, pop from the remainder to get the next pair.
                  (setf pattern (pop rest-pattern))
                  (setf data    (pop rest-data)))))
          ((and (listp pattern) (listp data))
           ;; We have two lists, one of which isn't nil.
           ;; Break it apart into bite-size chunks.
           (push (rest pattern) rest-pattern)
           (setf pattern (first pattern))
           (push (rest data) rest-data)
           (setf data (first data)))))))


;;; ******************************
;;; Examples ***********************
;;; ******************************
#|
* (unify '(a ?v (c e)) '(a b (?d e)))

((?D . C) (?V . B))
* (unify '(a ?v (c e)) '(a b (?d f)))

FAILED
* (recursive-unify '(?d ?c ?e) '(?c ?d ?c))

((?E . ?C) (?D . ?C))
* (recursive-unify '(?c ?d ?c) '(?d ?c ?e))

((?D . ?E) (?C . ?D))
* (unify '(?c ?d ?c) '(?d ?c ?e))

((?E . ?C) (?C . ?D) (?D . ?C))
* (unify '(?d ?c ?e) '(?c ?d ?c) nil t)

Pattern: (?D ?C ?E)
```

```
Data: (?C ?D ?C)
Environment: NIL
Pattern: ?D
Data: ?C
Environment: NIL
Pattern: NIL
Data: NIL
Environment: ((?C . ?D))
Pattern: (?C ?E)
Data: (?D ?C)
Environment: ((?C . ?D))
Pattern: ?C
Data: ?D
Environment: ((?C . ?D))
Pattern: NIL
Data: NIL
Environment: ((?D . ?C) (?C . ?D))
Pattern: (?E)
Data: (?C)
Environment: ((?D . ?C) (?C . ?D))
Pattern: ?E
Data: ?C
Environment: ((?D . ?C) (?C . ?D))
Pattern: ?E
Data: ?D
Environment: ((?D . ?C) (?C . ?D))
Pattern: ?E
Data: ?C
Environment: ((?D . ?C) (?C . ?D))
...
|#


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;; -*- Mode: LISP; Syntax: Common-lisp -*-
;;; Tue Aug  7 15:24:11 1990 by Mark Kantrowitz <mkant@GLINDA.OZ.CS.CMU.EDU>
;;; generic-search.lisp

;;; ************************************************************
;;; Generic Search ***********************************************
;;; ************************************************************
;;;
;;; This file implements a generic framework for search. It is intended
;;; as a pedagological tool for teaching students about the variety of
;;; forms of search as discussed in the AI literature.
;;;
;;; Written by Mark Kantrowitz, August 1990.
;;;
;;; Address: Carnegie Mellon University
;;;          School of Computer Science
;;;          Pittsburgh, PA 15213
;;;
```

```
;;; This code is in the public domain and is distributed without warranty
;;; of any kind.
;;;
;;; Portions of this code are based upon a problem set from MIT course 6.824.
;;;
;;; Use and copying of this software and preparation of derivative works
;;; based upon this software are permitted, so long as the following
;;; conditions are met:
;;;      o no fees or compensation are charged for use, copies, or
;;;        access to this software
;;;      o this copyright notice is included intact.
;;; This software is made available AS IS, and no warranty is made about
;;; the software or its performance.
;;;
;;; Please send bug reports, comments and suggestions to mkant@cs.cmu.edu.

;;; ************************************************************
;;; Documentation ********************************************
;;; ************************************************************
;;;
;;; When learning about some types of search commonly used in AI systems,
;;; it often helps to think in terms of a queue of nodes to be searched.
;;; Given a function which tests for the goal node, a function which
;;; finds the node's children, a function which dequeues a node for testing,
;;; and a function which merges the children into the queue, one can
;;; implement a wide variety of search functions. Comparing the functions
;;; used can help the student understand the difference between the
;;; various types of search.
;;;
;;; The function GENERIC-SEARCH below implements a generic framework
;;; for search by allowing the user to specify the functions describe above.
;;; It takes the following required arguments:
;;;    initial-state    the start node (the first state examined)
;;;    goal-p           a function to test whether a node satisfies the goal
;;;    children         a function which returns a list of a node's children
;;; and the following keyword arguments:
;;;    display-fn       a function which is called on each node as it is
;;;                     reached. useful to display the search progress
;;;    merge-fn         a function which returns a new queue when given a
;;;                     set of new nodes
;;;    dequeue-fn       a function which returns the next node off the queue.
;;; The global variable *search-queue* is accessible to each of these
;;; functions, and consists of a list representing the current search queue.
;;;
;;; The algorithm is quite simple. First it evaluates whether the current
;;; search state is a solution using GOAL-P. If not, it calls DISPLAY-FN
;;; on the node to display it. Then it uses CHILDREN to generate a set of
;;; child nodes and merges them into the search queue using MERGE-FN. It then
;;; calls DEQUEUE-FN to take the next state to be examined off the queue.
;;;
;;; This implementation makes no commitments about the representation of
;;; the search states or the search queue. The only requirement is that
;;; the MERGE-FN and DEQUEUE-FN functions use the same queue representation.
;;;
```

```
;;; Following the code for GENERIC-SEARCH, we list a variety of AI search
;;; techniques, along with the corresponding calls to generic search.
;;;

;;; ************************************************************
;;; Generic Search ********************************************
;;; ************************************************************
(defvar *search-queue* nil
  "The search queue is stored in this variable. The structure of the
   queue is determined by MERGE-FN and DEQUEUE-FN.")

(defun generic-search (initial-state goal-p children
                                    &key
                                    (display-fn #'print)
                                    (merge-fn #'(lambda (new-states)
                                                  (append new-states
                                                          *search-queue*)))
                                    (dequeue-fn #'(lambda ()
                                                    (pop *search-queue*))))
  "Generic search function. Arguments are an initial state and the functions:
      goal-p      -- tests whether a node satisfies the goal
      children    -- returns a list of a node's children
      display-fn  -- called on each node as it is reached
      merge-fn    -- returns a new queue when given a set of new nodes
      dequeue-fn  -- returns the next node off the queue
   *search-queue* contains the queue and is accessible to these functions."
  (let ((*search-queue* nil))
    (do ((current-state initial-state (funcall dequeue-fn)))
        ((funcall goal-p current-state) current-state)
      (funcall display-fn current-state)
      (setq *search-queue*
            (funcall merge-fn
                     (funcall children current-state))))))

(defvar *eval-fn* nil
  "This variable contains a function which, when applied to a node, returns
   a numeric evaluation of the node, such as the estimated remaining
   distance from the node.")


;;; ******************************
;;; Satisficing Paths **************
;;; ******************************
;;; The following types of search seek any path from the initial state to a
;;; goal state. The length of the discovered path is not important.

;;; *** Depth-First Search ***
;;; Add the children to the front of the queue.
#|
(generic-search initial-state goal-p children
                :merge-fn #'(lambda (new-states)
                              (append new-states *search-queue*))
                :dequeue-fn #'(lambda () (pop *search-queue*)))
|#
```

```
;;; *** Hill-Climbing ***
;;; Like depth-first search, but sorts the children by estimated remaining
;;; distance before adding them to the front of the queue.
#|
(generic-search initial-state goal-p children
                :merge-fn #'(lambda (new-states)
                              (append (sort new-states #'< :key *eval-fn*)
                                      *search-queue*))
                :dequeue-fn #'(lambda () (pop *search-queue*)))
|#

;;; *** Breadth-First Search ***
;;; Add the children to the end of the queue.
#|
(generic-search initial-state goal-p children
                :merge-fn #'(lambda (new-states)
                              (append *search-queue* new-states))
                :dequeue-fn #'(lambda () (pop *search-queue*)))
|#

;;; *** Beam Search ***
;;; Like breadth-first search, but keeps only the k best nodes at each level.
#|
(generic-search initial-state goal-p children
                :merge-fn #'(lambda (new-states)
                              (append *search-queue* new-states))
                :dequeue-fn #'(lambda ()
                                (let ((node (pop *search-queue*)))
                                  (if (eq node '*q-tag*)
                                      (append (first (sort *search-queue*
                                                           #'< :key *eval-fn*)
                                                     *k*)
                                              '*q-tag*)
                                      node))))
|#

;;; *** Best-First Search ***
;;; The next node searched is the best node, no matter where it is in the
;;; tree. Sorts the entire queue by the estimated remaining distance after
;;; adding children.
#|
(generic-search initial-state goal-p children
                :merge-fn #'(lambda (new-states)
                              (sort (append new-states *search-queue*)
                                    #'< :key *eval-fn*))
                :dequeue-fn #'(lambda () (pop *search-queue*)))
|#


;;; ******************************
;;; Optimal Paths *****************
;;; ******************************
;;; Finds the shortest (optimal) path to the goal node.
```

```
;;; *** Branch and Bound ***
;;; Extends the shortest (least cost) partial path to the goal. Sorts the
;;; queue by accumulated cost so far (least cost in front) after adding
;;; children to queue. Looks like best-first search, except the *eval-fn*
;;; is different.

;;; *** Branch and Bound with Underestimates ***
;;; Instead of using accumulated cost so far, it adds an underestimate
;;; (lower bound) on the remaining distance to the total distance already
;;; travelled to obtain an underestimate of the total path length. It uses
;;; this underestimate of the total path length to sort the queue.

;;; *** Dynamic Programming ***
;;; Keeps a table of the best path to each node. Discards redundant paths.

;;; *** A* Search ***
;;; Branch and Bound with Underestimates and Dynamic Programming.

;;; ******************************
;;; Priority Queue ****************
;;; ******************************
;;; The functions priority-merge-fn and priority-dequeue-fn use a different
;;; structure for the search queue, and implement a priority queue (best-first
;;; search), where the entries in the queue are (value . node) pairs.

(defmacro splice (state queue)
  `(let* ((value (funcall *eval-fn* ,state))
          (node (cons value ,state)))
     (do ((qp ,queue (cdr qp))
          (oqp nil qp))
         ((or (null qp) (> value (car (first qp))))
          (if oqp
              (rplacd oqp
                      (cons node (cdr oqp)))
              (push node ,queue))))))

(defun priority-merge-fn (new-states)
  "Maintains a priority-queue of (priority . state) values in descending
   order of priority (useful for best-first, A*, etc.)."
  (let ((queue *search-queue*))
    (dolist (state new-states)
      (splice state queue))
    queue))

(defun priority-dequeue-fn ()
  "Partner for priority-merge-fn.  Dequeues top (pri . state) value and
   returns the state."
  (cdr (pop *search-queue*)))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;  PARCIL - A Parser for C syntax In Lisp
;;;  version 0.1a
;;;
;;;  copyright (c) 1992 by Erann Gat, all rights reserved
;;;
;;;  This program is free software; you can redistribute it and/or modify
;;;  it under the terms of the GNU General Public License as published by
;;;  the Free Software Foundation.
;;;
;;;  This program is distributed in the hope that it will be useful,
;;;  but WITHOUT ANY WARRANTY; without even the implied warranty of
;;;  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
;;;  GNU General Public License for more details.
;;;
;;;  You should have received a copy of the GNU General Public License
;;;  along with this program; if not, write to the Free Software
;;;  Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
;;;
;;;
;;;  This is a very preliminary release and almost certainly contains bugs.
;;;  Please send bug reports and comments to:
;;;  Erann Gat
;;;  JPL MS 525-3660
;;;  4800 Oak Grove Drive
;;;  Pasadena, CA 91109
;;;  (818) 306-6176
;;;  gat@robotics.jpl.nasa.gov or gat@aig.jpl.nasa.gov
;;;
;;;  Revision history:
;;;  v0.1a - Initial release
;;;

;;;  PARCIL is a parser for a subset of the syntax for the C programming
;;;  language.  PARCIL is written in Common Lisp, making it potentially
;;;  a useful building block for user interfaces for people who do not
;;;  like prefix syntax.
;;;
#|
PARCIL is a recursive descent parser optimized to parse C.  This makes it
fairly brittle and difficult to modify.  However, it does make it fairly fast,
and it also allows the parser to deal with lots of C idiosyncrasies which are
difficult to implement in general-purpose parsers, e.g. operator precedence,
prefix and postfix operators, etc.
  NOTE:  While PARCIL is designed to be a component in user interfaces for
people who are not regular LISP users, it is probably not usable for that
purpose as-is.  There are two major problems with it.  First, it is
incomplete. It currently includes no support for any high-level C construct
(i.e. it implements the syntax described in the original Kernighan and Richie
book, section 18.1).  The second problem is that PARCIL is so faithful to C
syntax that it can easily fool the unwary into believing that they are writing
```

```
C code when in fact they are writing LISP code, only with a different syntax.
You need a fairly deep understanding of the distinction between syntax and
semantics in order to use PARCIL.  The main stumbling block to its use by
beginners is that PARCIL does very little error checking.  Thus, many errors
which should be detected by PARCIL are passed on and caught by LISP.  The
resulting error messages can be very cryptic if you don't know what's going
on.
  PHILOSOPHICAL RANT:  Infix notation is a blight on the intellectual
landscape.  It is confusing to read, difficult to parse, and to avoid
ambiguity must rely on precedence rules that are hopelessly obscure.  People
who prefer infix notation do so only because they have been indoctrinated to
it since childhood and do not have the intellectual strength to break free.
It is far better to convince people to use prefix notation, with its easy to
read and easy to parse, unambiguous syntax, than to provide them with crutches
such as PARCIL which perpetuate such evils as infix, prefix and postfix unary
operators.  (In C, "x++++++++y" is a legal expression, and the first *
doesn't mean the same thing as all the other *'s.)  Nevertheless, I
acknowledge the reality that infix and C are here to stay, and that is why I
have written PARCIL.  But that doesn't mean I have to like it.
|#
;;;
;;;  USER'S GUIDE:
;;;
;;;  The top-level function is called PARCIL.  Pass a string consisting of a C
;;;  expression (not a command!) to PARCIL and it will return a parsed ;;;
;;;  version. For example:
;;;
;;;  (parcil "x=y*sin(pi/2.7)") ==> (SETF X (* Y (SIN (/ PI 2.7))))
;;;
;;;  PARCIL supports all syntax defined in section 18.1 of the original
;;;  Kernighan and Ritchie book, plus all C numerical syntax including floats
;;;  and radix syntax (i.e. 0xnnn, 0bnnn, and 0onnn).  In addition, PARCIL
;;;  supports multiple array subscripts.  There is also a preliminary version
;;;  of {} blocks, but it doesn't quite do the right thing.  Parcil also
;;;  allows strings to be delimited using single quotes as well as double
;;;  quotes (but you must use the same type to close the string as you did to
;;;  open it).

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;  Program starts here
;;;

;;;  Misc. utilities
;;;
(defmacro iterate (name args &rest body)
  `(labels ((,name ,(mapcar #'car args) ,@body))
     (,name ,@(mapcar #'cadr args))))

(defmacro while (condition &body body)
  `(iterate loop () (if ,condition (progn ,@body (loop)))))

;;;  Crufty pseudo-text-file interface.  Don't let impressionable young minds
;;;  see this code.
```

```
;;;
(defvar *the-string* "")
(defvar *the-pointer* 0)

(defun parse-init (s)
  (setq *the-string* s)
  (setq *the-pointer* 0))

(defun eof? (&optional (offset 0))
  (>= (+ *the-pointer* offset) (length *the-string*)))

(defun peek (&optional (offset 0))
  (if (eof? offset)
    nil
    (char *the-string* (+ *the-pointer* offset))))

(defun readc ()
  (prog1 (peek) (incf *the-pointer*)))

;;;  The PARCIL tokenizer.  (FSA?  What's an FSA?)
;;;
(defun letter? (c)
  (and (characterp c) (or (char<= #\a c #\z) (char<= #\A c #\Z))))

(defun digit? (c)
  (and (characterp c) (char<= #\0 c #\9)))

(defun ident? (thing)
  (and thing
       (symbolp thing)
       (letter? (char (symbol-name thing) 0))))

(defvar *binary-ops*
  '((\. ->) (* / %) (+ -) (<< >>) (< > <= >=) (== !=) (&) (^) (\|) (&&) (\|\|)
    (= += -= *= /= %= &= ^= \|= >>= <<=)))

;;; Any binary operator in this alist will be renamed in the parsed version.
(defvar *binop-alist*
  '((\. . struct-ref) (= . setf) (% . mod) (<< . ashl) (>> . ashr)
    (& . logand) (^ . logxor) (\| . logior) (&& . and) (\|\| . or)))

(defun binop? (s) (member s *binary-ops* :test #'member))
(defun assignop? (s) (member s (car (last *binary-ops*))))
(defun priority (s)
  (let ( (p (position s *binary-ops* :test #'member)) )
    (and p (- 40 p))))
(defun translate-binop (op) (or (cdr (assoc op *binop-alist*)) op))

(defun eat-spaces ()
  (do () ( (not (eql (peek) #\Space)) )
    (readc)))

(defun syntax-error ()
  (error "Syntax error near ~S"
```

```
          (subseq *the-string* (max 0 (1- *the-pointer*)))))

(defun parse-fixnum (&optional (base 10))
  (multiple-value-bind
    (n cnt) (parse-integer *the-string* :start *the-pointer* :radix base
                           :junk-allowed t)
    (setq *the-pointer* cnt)
    (if (not (numberp n)) (syntax-error))
    n))

(defun parse-atom ()
  (eat-spaces)
  (if (eof?)
    nil
    (let ( (c (peek)) )
      (cond ( (letter? c) (parse-symbol) )
            ( (eql c #\0) (if (letter? (peek 1))
                              (parse-radix-integer)
                              (parse-number)) )
            ( (digit? c) (parse-number) )
            ( (or (eql c #\") (eql c #\')) (parse-string c) )
            (t (parse-operator))))))

(defun parse-symbol ()
  (intern
   (string-upcase
    (with-output-to-string (s)
      (while (let ( (c (peek)) )
                  (and c (or (letter? c) (digit? c) (eql c #\_))))
        (princ (readc) s))
      s))))

(defun parse-radix-integer ()
  (readc)
  (parse-fixnum (ecase (readc) (#\x 16) (#\o 8) (#\b 2))))

(defun parse-number ()
  (let* ( (n1 (parse-fixnum))
          (c (peek)) )
    (prog ( (d 0.1) )
      (if (eql c #\.) (go decimal))
      (if (or (eql c #\e) (eql c #\E)) (go expt))
      (return n1)
    decimal
      (readc)
      (let ( (c (peek)) )
        (when (digit? c)
          (incf n1 (* d (- (char-code c) (char-code #\0))))
          (setf d (/ d 10))
          (go decimal))
        (if (or (eql c #\e) (eql c #\E)) (go expt))
        (return n1))
    expt
      (readc)
```

```
      (let ( (e (parse-fixnum)) )
        (return (* n1 (expt 10 e)))))))

(defun parse-string (terminator)
  (readc)
  (with-output-to-string (s)
    (iterate loop ()
      (let ( (c (readc)) )
        (when (eql c terminator) (return-from loop s))
        (princ c s)
        (loop)))))

(defun parse-operator ()
  (let* ( (c (intern (string (readc))))
          (s (intern (format nil "~A~A" c (peek)))) )
    (cond ( (member s '(<< >>))
            (readc)
            (if (eql (peek) #\=)
              (intern (format nil "~A~A" s (readc)))
              s) )
          ( (member s '(++ -- << >> -> <= >= != == &&
                        += -= *= /= %= &= ^= \|= \|\|))
            (readc)
            s )
          (t c))))

;;;  Crufty interface to the tokenizer.
;;;
(defvar *next*)

(defun scan ()
  (setf *next* (parse-atom)))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;;  The recursive-descent parser.  Look Ma, no tables!
;;;
(defun parse-expression (&optional (priority -1))
  (iterate loop ( (result (parse-term)) )
    (let ( (op (translate-binop *next*))
           (new-priority (priority *next*)) )
      (cond
       ( (assignop? *next*) (scan) (list op result (loop (parse-term))) )
       ( (and (binop? *next*) (> new-priority priority))
         (scan) (loop (list op result (parse-expression new-priority))) )
       (t result)))))

(defun parse-arglist (&optional (terminator '\)) (separator '\,))
  (iterate loop ()
    (cond ( (null *next*) (error "Missing ~S" terminator) )
          ( (eq *next* terminator) (scan) nil )
          (t (let ( (arg1 (parse-expression)) )
               (unless (or (eq *next* separator) (eq *next* terminator))
                 (syntax-error))
```

```
                (if (eq *next* separator) (scan))
                (cons arg1 (loop)))))))

;;;  Any prefix unary operator included in this table will be renamed in the
;;;  parsed version. (Postfix ++ and -- are handled specially, in PARSE-TERM.)
(defvar *unary-op-alist*
  '((* . deref) (& . address-of)
    (- . -) (! . not) (~ . lognot)
    (++ . incf) (-- . decf)))

;;;  This function parses what K&R call primary expressions.  These include
;;;  numbers, variables, structure references, array references, and all unary
;;;  operators.  Parsing of curly brackets is also stuck in here, though it
;;;  probably shouldn't be.  The weird precedence rules make this a fairly
;;;  hairy and brittle piece of code.
;;;
(defun parse-term ()
  (iterate loop ( (term (prog1 *next* (scan))) )
    (cond
     ( (numberp term) term )
     ( (assoc term *unary-op-alist*)
       (list (cdr (assoc term *unary-op-alist*)) (parse-term)) )
     ( (eq term '\( )
       (cons 'progn (parse-arglist)) )
     ( (eq term '{)
       (list* 'let '() (parse-arglist '} '\;)) )
     ( (eq *next* '\( )
       (scan)
       (loop (cons term (parse-arglist))) )
     ( (eq *next* '\[ )
       (scan)
       (loop `(aref ,term ,@(parse-arglist ']))) )
     ( (eq *next* '\.)
       (loop `(struct-ref ,term ,(prog1 (scan) (scan)))) )
     ( (eq *next* '->)
       (loop `(-> ,term ,(prog1 (scan) (scan)))) )
     ( (eq *next* '++)
       (scan)
       (loop `(prog1 ,term (incf ,term))) )
     ( (eq *next* '--)
       (scan)
       (loop `(prog1 ,term (decf ,term))) )
     (t
      (if (and (atom term) (not (ident? term)))
        (syntax-error))
      term))))

;;;;;;;;;;;;;;;;
;;;
;;;  The top level
;;;
(defun parcil (s)
  (parse-init s)
  (scan)
```

```
  (prog1
    (parse-expression)
    (if *next* (syntax-error))))
                ; If there's stuff left over something went wrong.

;;; Reader hook (optional)
;;; Evaluating the following forms will allow you to type C-like expressions
;;; directly at the lisp reader and have them evaluated, e.g.:
;;;
;;;  ? #{ ( x=1,y=2,print(x+y),sin(pi/2) ) }
;;;  3
;;;  1.0
;;;  ?
(defun |#{-reader| (stream char arg)
  (declare (ignore char arg))
  (parcil
   (with-output-to-string (s)
     (loop
       (let ( (c (read-char stream)) )
         (if (eql c #\})
           (return s)
           (princ c s)))))))

(set-dispatch-macro-character #\# #\{ #'|#{-reader|)
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; GPS engine for blocks world
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


(defvar *dbg-ids* nil "Identifiers used by dbg")

(defvar *state* nil "The current state: a list of conditions.")

(defvar *ops* nil "A list of available operators.")

(defstruct op "An operation"
  (action nil) (preconds nil) (add-list nil) (del-list nil))

(defun find-all (item sequence &rest keyword-args
                 &key (test #'eql) test-not &allow-other-keys)
  "Find all those elements of sequence that match item,
  according to the keywords.  Doesn't alter sequence."
  (if test-not
      (apply #'remove item sequence
             :test-not (complement test-not) keyword-args)
      (apply #'remove item sequence
             :test (complement test) keyword-args)))

(defun member-equal (item list)
  (member item list :test #'equal))

;;; ==============================

(defun executing-p (x)
  "Is x of the form: (executing ...) ?"
  (starts-with x 'executing))

(defun starts-with (list x)
  "Is this a list whose first element is x?"
  (and (consp list) (eql (first list) x)))

(defun convert-op (op)
  "Make op conform to the (EXECUTING op) convention."
  (unless (some #'executing-p (op-add-list op))
    (push (list 'executing (op-action op)) (op-add-list op)))
  op)

(defun op (action &key preconds add-list del-list)
  "Make a new operator that obeys the (EXECUTING op) convention."
  (convert-op
    (make-op :action action :preconds preconds
             :add-list add-list :del-list del-list)))

;;; ==============================


(defun apply-op (state goal op goal-stack)
```

```
  "Return a new, transformed state if op is applicable."
  (dbg-indent :gps (length goal-stack) "Consider: ~a" (op-action op))
  (let ((state2 (achieve-all state (op-preconds op)
                             (cons goal goal-stack))))
    (unless (null state2)
      ;; Return an updated state
      (dbg-indent :gps (length goal-stack) "Action: ~a" (op-action op))
      (append (remove-if #'(lambda (x)
                             (member-equal x (op-del-list op)))
                         state2)
              (op-add-list op)))))

(defun appropriate-p (goal op)
  "An op is appropriate to a goal if it is in its add list."
  (member-equal goal (op-add-list op)))

;;; ==============================

(defun use (oplist)
  "Use oplist as the default list of operators."
  ;; Return something useful, but not too verbose:
  ;; the number of operators.
  (length (setf *ops* oplist)))

;;; ==============================

(defun GPS (state goals &optional (*ops* *ops*))
  "General Problem Solver: from state, achieve goals using *ops*."
  (remove-if-not #'action-p
                 (achieve-all (cons '(start) state) goals nil)))

(defun action-p (x)
  "Is x something that is (start) or (executing ...)?"
  (or (equal x '(start)) (executing-p x)))



(defun make-block-ops (blocks)
  (let ((ops nil))
    (dolist (a blocks)
      (dolist (b blocks)
        (unless (equal a b)
          (dolist (c blocks)
            (unless (or (equal c a) (equal c b))
              (push (move-op a b c) ops)))
          (push (move-op a 'table b) ops)
          (push (move-op a b 'table) ops))))
    ops))

(defun move-op (a b c)
  "Make an operator to move A from B to C."
  (op `(move ,a from ,b to ,c)
      :preconds `((space on ,a) (space on ,c) (,a on ,b))
      :add-list (move-ons a b c)
```

```
      :del-list (move-ons a c b)))

(defun move-ons (a b c)
  (if (eq b 'table)
      `((,a on ,c))
      `((,a on ,c) (space on ,b))))


;;; ==============================

(defun achieve-all (state goals goal-stack)
  "Achieve each goal, trying several orderings."
  (some #'(lambda (goals) (achieve-each state goals goal-stack))
        (orderings goals)))

(defun achieve-each (state goals goal-stack)
  "Achieve each goal, and make sure they still hold at the end."
  (let ((current-state state))
    (if (and (every #'(lambda (g)
                        (setf current-state
                              (achieve current-state g goal-stack)))
                    goals)
             (subsetp goals current-state :test #'equal))
        current-state)))

(defun orderings (l)
  (if (> (length l) 1)
      (list l (reverse l))
      (list l)))

;;; ==============================

(defun achieve (state goal goal-stack)
  "A goal is achieved if it already holds,
  or if there is an appropriate op for it that is applicable."
  (dbg-indent :gps (length goal-stack) "Goal: ~a" goal)
  (cond ((member-equal goal state) state)
        ((member-equal goal goal-stack) nil)
        (t (some #'(lambda (op) (apply-op state goal op goal-stack))
                 (appropriate-ops goal state))))) ;***

(defun appropriate-ops (goal state)
  "Return a list of appropriate operators,
  sorted by the number of unfulfilled preconditions."
  (sort (copy-list (find-all goal *ops* :test #'appropriate-p)) #'<
        :key #'(lambda (op)
                 (count-if #'(lambda (precond)
                               (not (member-equal precond state)))
                           (op-preconds op)))))


;;;; The Debugging Output Facility:

(defvar *dbg-ids* nil "Identifiers used by dbg")
```

```
(defun dbg (id format-string &rest args)
  "Print debugging info if (DEBUG ID) has been specified."
  (when (member id *dbg-ids*)
    (fresh-line *debug-io*)
    (apply #'format *debug-io* format-string args)))

(defun debug (&rest ids)
  "Start dbg output on the given ids."
  (setf *dbg-ids* (union ids *dbg-ids*)))

(defun undebug (&rest ids)
  "Stop dbg on the ids.  With no ids, stop dbg altogether."
  (setf *dbg-ids* (if (null ids) nil
                      (set-difference *dbg-ids* ids))))

(defun dbg-indent (id indent format-string &rest args)
  "Print indented debugging info if (DEBUG ID) has been specified."
  (when (member id *dbg-ids*)
    (fresh-line *debug-io*)
    (dotimes (i indent) (princ "  " *debug-io*))
    (apply #'format *debug-io* format-string args)))



;;;

#|
("The Blocks World Domain")
  "Another domain that has attracted more than its share of attention in AI"
  "circles is the blocks world domain."
  ((use (make-block-ops '(a b))) => 4)
  "The simplest possible problem is stacking one block on another."
  ((gps '((a on table) (b on table) (space on a) (space on b)
          (space on table))
        '((a on b) (b on table))) =>
        ((START)
       (EXECUTING (MOVE A FROM TABLE TO B))))
  "Here is a slightly more complex problem: inverting a stack of two blocks."
  "This time we show the debugging output:"
  ((debug :gps) )
  ((gps '((a on b) (b on table) (space on a) (space on table))
        '((b on a))) =>
        ((START)
       (EXECUTING (MOVE A FROM B TO TABLE))
       (EXECUTING (MOVE B FROM TABLE TO A))))
  ((undebug))
  "Now we move on to the three block world."
  ((use (make-block-ops '(a b c))) => 18)
  "We try some problems:"
  ((gps '((a on b) (b on c) (c on table) (space on a) (space on table))
        '((b on a) (c on b))) =>
        ((START)
       (EXECUTING (MOVE A FROM B TO TABLE))
```

```
    (EXECUTING (MOVE B FROM C TO A))
    (EXECUTING (MOVE C FROM TABLE TO B))))
((gps '((c on a) (a on table) (b on table)
       (space on c) (space on b) (space on table))
     '((c on table) (a on b))) =>
     ((START)
    (EXECUTING (MOVE C FROM A TO TABLE))
    (EXECUTING (MOVE A FROM TABLE TO B))))
((gps '((a on b) (b on c) (c on table) (space on a) (space on table))
    '((b on a) (c on b)))  =>
    ((START)
     (EXECUTING (MOVE A FROM B TO TABLE))
     (EXECUTING (MOVE B FROM C TO A))
     (EXECUTING (MOVE C FROM TABLE TO B))))

((gps '((a on b) (b on c) (c on table) (space on a) (space on table))
    '((c on b) (b on a))) =>
    ((START)
     (EXECUTING (MOVE A FROM B TO TABLE))
     (EXECUTING (MOVE B FROM C TO A))
     (EXECUTING (MOVE C FROM TABLE TO B))))
"The Sussman Anomaly"
((setf start '((c on a) (a on table) (b on table) (space on c)
              (space on b) (space on table))) )
((gps start '((a on b) (b on c))) => NIL)
((gps start '((b on c) (a on b))) => NIL)

|#
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; STREAMS with DELAYED EVALUATION
;;;
;;;  Ideas from Pavel Curtis, Richard Waters, Guy Steele and
;;;    the Common LISP community.  Most of the code by
;;;    George Lugar and William Stubblefield.
;;;
;;;  DELAY and FORCE permit fine grain program control of when
;;;  a function or subprogram is executed.  Instead of flow of
;;;  control being managed by glue logic, loops, and function
;;;  nesting, evaluation is either delayed or called when needed
;;;  by the dynamic context of the program.
;;;
;;;  Conceptually, a STREAM is a dynamically available data
;;;  structure, implemented by a GENERATOR function:
;;;
;;;  {first-data-item <generator-function-for-next-data-item>}
;;;
;;;  When the first item is used by a process, the generator is
;;;  called to construct the next data-item.
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


(defmacro delay (exp) `(function (lambda () ,exp)))
(defun force (function-closure) (funcall function-closure))

;;;add a new first element to a stream
(defmacro cons-stream (new stream)
  `(cons ,new (delay ,stream)))

;;;get the first element
(defun first-stream (stream)
  (car stream))

;;;the rest of the stream is a generator function
(defun rest-stream (stream)
  (force (rest stream)))

;;;test for empty stream
(defun empty-streamp (stream)
  (null stream))

;;;make an empty stream
(defun make-empty-stream ()
  nil)

;;;append two streams
(defun combine-streams (s1 s2)
  (append s1 s2))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;GENERIC-FILTER for LISTS
```

```
(defun filter (lst test)
  (cond ((null lst) nil)
        ((funcall test (first lst))
         (cons (first lst) (filter (rest lst) test)))
        (T (filter (rest lst) test))))

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;GENERIC-FILTER for STREAMS

(defun filter-stream (stream test)
  (cond ((empty-streamp stream) (make-empty-stream))
        ((funcall test (first-stream stream))
         (cons-stream (first-stream stream)
                      (filter-stream (rest-stream stream) test)))
        (T (filter-stream (rest-stream stream) test))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;FIBONACCI STREAM FILTER
;;;   consider generating the first N odd Fibonacci numbers
;;;   using the minimum of computational effort
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;the standard mathematical definition of Fibonacci numbers
;;;  fib[1] = 0
;;;  fib[2] = 1
;;;  fib[n] = fib[n-1] + fib[n-2]
(defun fib (n)
  (cond ((= n 1) 0)
        ((= n 2) 1)
        (T (+ (fib (- n 1)) (fib (- n 2))))))

(defun collect-fibs (n)
  (let ((result nil))
    (dotimes (i n (reverse result))  ;iterator i starts at 0, so we must
      (push (fib (+ i 1)) result)))) ;add 1 to start fib counter at 1


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;an efficient doloop implementation which collects prior results
(defun memo-fibs (n)
  (let ((memo '(1 0)))
    (dotimes (i (- n 2))
      (push (+ (first memo) (second memo)) memo))
    (reverse memo)))

;;;a memoized doloop with an arbitrary filter
(defun filter-fibs (n filter-test-fn termination-test-fn)
  (do ((acc '(1 0))
       (test-acc nil))
      ((funcall termination-test-fn test-acc n)
       (reverse test-acc))
    (let ((new (+ (first acc) (second acc))))
```

```
      (push new acc)
      (when (funcall filter-test-fn new)
        (push new test-acc)))))

#|
;;; code fitted with lots of debug options,
;;;   to illustrate inline debug tools
(defun filter-fibs (n filter-test-fn termination-test-fn)
  (do ((acc '(1 0))
       (test-acc nil))
      ((funcall termination-test-fn test-acc n)
;(print (list 'VALUES-PRIOR-TO-DO-EXIT acc test-acc n))
;(break "At exit: count ~A fibs ~A filtered-fibs ~A" acc test-acc n)
       (reverse test-acc))
    (let ((new (+ (first acc) (second acc))))
;(break "new fib value: ~A"  new)
      (push new acc)
;(print (list 'new-stack acc))
      (when (funcall filter-test-fn new)
        (push new test-acc)))))
|#

(defun have-sufficient (lst n)
  (= (length lst) n))

(defun three-fives (n)
  (at-least-digits n 3 5))

;;; given a number N, find at least THIS-MANY of a given digit DIGIT
(defun at-least-digits (n this-many digit)
  (let ((digit-string (format nil "~D" n)) ;convert symbol to string
        (test-char (digit-char digit)))    ;convert symbol to character
    (count-digits-test digit-string this-many test-char)))

(defun count-digits-test (string num char)
  (>= (count char string) num))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;an implementation based on streams
;;;  cleaner, more maintainable, and as efficient as FILTER-FIBS
;;;  better than FILTER-FIBS when the termination test is complex
(defun fibonacci-stream ()
  (fib-stream-aux 0 1))

(defun fib-stream-aux (fib1 fib2)
  (cons-stream (+ fib1 fib2)
               (fib-stream-aux fib2 (+ fib1 fib2))))

(defun accumulate (n stream)
  (cond ((zerop n) nil)
        (T (cons (first-stream stream)
                 (accumulate (- n 1) (rest-stream stream))))))
```

```
(defun filter-stream-fibs (n test-fn)
  (accumulate n (filter-stream (fibonacci-stream) test-fn)))


;;;most abstract
(defun accumulate-filtered-stream (number-items generator-fn filter-fn)
  (accumulate number-items
    (filter-stream (funcall generator-fn)  filter-fn)))


#|
;;;filter examples
(filter '(1 3 -9 5 -2 -7 6) #'plusp)
;;==>  (1 3 5 6)
(filter '(1 2 3 4 5 6 7 8 9) #'evenp)
;;==>  (2 4 6 8)
(filter '(1 a b 3 c 4 7 d) #'numberp)
;;==>  (1 3 4 7)

;;;try these
(filter (collect-fibs 20) #'oddp)    ;yields only 13 odd fibs, so guess
(filter (collect-fibs 30) #'oddp)    ;to get 20 odd fibs
(time (filter (collect-fibs 30) #'oddp))
;;;don't try (filter (collect-fibs <some-guess>) #'three-fives)

;;;try these
(filter-fibs 20 #'oddp #'have-sufficient)
(filter-fibs 8 #'three-fives #'have-sufficient)
(time (dotimes (i 10) (filter-fibs 8 #'three-fives #'have-sufficient)))

;;;try these
(filter-stream-fibs 20 #'oddp)
(filter-stream-fibs 8 #'three-fives)
(time (dotimes (i 10) (filter-stream-fibs 8 #'three-fives)))

;;;try this
(accumulate-filtered-stream 8 #'fibonacci-stream #'three-fives)
|#
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;; LOGIC PROGRAMMING INTERPRETER
;;;
;;;  from George Lugar and William Stubblefield
;;;    Artificial Intelligence and the Design of Expert Systems
;;;    Benjamin/Cummings, 1989
;;;  Additional examples by William Bricken
;;;
;;;  Takes a database of horn clauses and unifies variables
;;;    to answer queries
;;;
;;;  Streams simulated by lists, does not implement generators.

(defun first-stream (stream)  (car stream))
(defun rest-stream (stream)  (cdr stream))
(defun cons-stream (exp stream)  (cons exp stream))
(defun combine-streams (s1 s2)  (append s1 s2))
(defun empty-streamp (stream)  (null stream))
(defun make-empty-stream ()  nil)

(defvar *ASSERTIONS* nil)

;;;top level interpreter loop
(defun logic-shell ()
  (print 'logic>)             ;the prompt
  (let* ((goal-string (read-line))
         (goal (read-from-string goal-string)))
    (cond ((member goal '(QUIT EXIT BYE)) '*EXITING-LOGIC-SHELL*)
          (T (let ((soln (solve goal nil)))
;(pprint (list 'SOLUTION soln))
             (print-solutions goal soln)
             (terpri)
             (logic-shell))))))

;;;SOLVE takes a goal and a set of substitutions and
;;; returns a stream of augmented substitutions which satisfy the goal
(defun solve (goal subs)
  (cond ((conjunctive-goalp goal)
         (filter-thru-conj-goals
          (body goal) (cons-stream subs (make-empty-stream))))
        (T (infer goal subs *ASSERTIONS*))))

;;;FILTER-THRU-CONJ-GOALS takes a list of goals and a stream of
;;;  substitutions and filters through the goals, eliminating failures
(defun filter-thru-conj-goals (goals subst-stream)
  (cond ((null goals) subst-stream)
        ((filter-thru-conj-goals
          (rest goals)
          (filter-thru-goal (first goals) subst-stream)))))

(defun filter-thru-goal (goal subst-stream)
  ;(declare (notinline filter-thru-goal))
  (cond ((empty-streamp subst-stream)
```

```
         (make-empty-stream))
        (T (combine-streams
            (solve goal (first-stream subst-stream))
            (filter-thru-goal goal (rest-stream subst-stream))))))

;;;INFER attempts to infer a goal from a knowledge-base
(defun infer (goal subs kb)
  ;(declare (notinline infer))
  (cond ((null kb) (make-empty-stream))
        (T (let ((assertion (rename-variables (car kb))))
             (combine-streams
              (cond ((rulep assertion)
                     (use-rule goal assertion subs))
                    (T (use-fact goal assertion subs)))
              (infer goal subs (rest kb)))))))

;;;USE-RULE attempts to infer a goal from a rule by backward chaining
(defun use-rule (goal rule subs)
  (let ((match (unify goal (conclusion rule) subs)))
    (cond ((equal match '*FAILED*) (make-empty-stream))
          (T (solve (premise rule) match)))))

;;;USE-FACT attempts to match a goal with a fact
(defun use-fact (goal pat subs)
  (let ((match (unify goal pat subs)))
    (cond ((equal match '*FAILED*) (make-empty-stream))
          (T
;(pprint (list 'MATCH= match))
           (cons-stream match (make-empty-stream))))))


;;;print utility
(defun print-solutions (goal sub-stream)
  ;(declare (notinline print-solutions))
  (cond ((empty-streamp sub-stream) (print '*DONE*))
        (T (print (apply-substitutions goal (first-stream sub-stream)))
           (print-solutions goal (rest-stream sub-stream)))))

(defun apply-substitutions (pat slst)
  ;(declare (notinline apply-substitutions))
  (cond ((is-constantp pat) pat)
        ((varp pat)
         (let ((binding (get-binding pat slst)))
           (cond (binding
                  (apply-substitutions (get-binding-value binding) slst))
                 (T pat))))
        (T (cons (apply-substitutions (first pat) slst)
                 (apply-substitutions (rest pat) slst)))))


;;; rule format is
;;;  (RULE IF <premise> THEN <conclusion>)

(defun premise (rule)
```

```
  (nth 2 rule))

(defun conclusion (rule)
  (nth 4 rule))

(defun rulep (pat)
  (and (listp pat) (equal (nth 0 pat) 'RULE)))

;;; conjunctive goals are in the form
;;;  (AND <goal1> <goal2> ... <goalN>)

(defun conjunctive-goalp (goal)
  (and (listp goal) (equal (first goal) 'AND)))

(defun body (goal)
  (rest goal))

;;;rename variables each time a rule is used

(defvar *name-list* nil)

(defun rename-variables (assertion)
  (setq *name-list* nil)
  (rename-recur assertion))

(defun rename-recur (exp)
  (cond ((is-constantp exp) exp)
        ((varp exp) (rename exp))
        (T (cons (rename-recur (first exp))
                 (rename-recur (rest exp))))))

(defun rename (var)
  (list 'VAR
        (or (cdr (assoc (second var) *name-list*))
            (let ((name (gensym)))
              (setq *name-list* (acons (second var) name *name-list*))
              name))))

;;;UNIFY utility
(defun unify (p1 p2 slst)
  (cond ((equal slst '*FAILED*) '*FAILED*)
        ((varp p1) (match-var p1 p2 slst))
        ((varp p2) (match-var p2 p1 slst))
        ((is-constantp p1)
         (if (equal p1 p2) slst '*FAILED*))
        ((is-constantp p2) '*FAILED*)
        (T (unify (rest p1) (rest p2)
                  (unify (first p1) (first p2) slst)))))

(defun match-var (var pat slst)
  (cond ((equal var pat) slst)
        (T (let ((binding (get-binding var slst)))
             (cond (binding
                    (unify (get-binding-value binding) pat slst))
```

```
                  ((occursp var pat) '*FAILED*)
                  (T (add-substitution var pat slst)))))))

(defun occursp (var pat)
  (cond ((equal var pat) T)
        ((or (varp pat) (is-constantp pat)) nil)
        (T (or (occursp var (first pat))
               (occursp var (rest pat))))))

(defun is-constantp (item)
  (atom item))

(defun varp (item)
  (and (listp item)
       (equal (length item) 2)
       (equal (first item) 'VAR)))

(defun get-binding (var slst)
  (assoc var slst :test #'equal))

(defun get-binding-value (binding)
  (cdr binding))

(defun add-substitution (var pat slst)
  (acons var pat slst))



#|
;;; After the code is loaded, select an *ASSERTION* database
;;; and load it into the LISP environment.  **Do not** try to
;;; load or evaluate functions in the LOGIC-SHELL. it is a
;;; continuous loop which hides the LISP context and evaluation
;;; mechanisms.

;;;liking example
;;;  a query would look like (bill likes (var x))
;;;try  (george likes (var x))
;;;try  ((var x) likes kids)
;;;try  ((var x) (var y) everything)
;;;try  (bill friend-of (var x))    and explain the result


(setq *assertions*
      '(
        (george likes beer)
        (george likes kate)
        (george likes kids)
        (tom likes everything)
        (bill likes kids)
        (bill likes music)
        (bill likes pizza)
        (bill likes wine)
```

```
        (rule if
              (and ((var x) likes (var y))
                   ((var z) likes (var y)))
              then
              ((var x) friend-of (var z)))
        ))

;;; new liking example
;;;  facts which declare the uniqueness of names have been added
;;;  and the FRIEND-OF rule now excludes being your own friend
;;;try  (bill friend-of (var x))

(setq *assertions*
      '(
        (george likes beer)
        (george likes kate)
        (george likes kids)
        (tom likes everything)
        (bill likes kids)
        (bill likes music)
        (bill likes pizza)
        (bill likes wine)
        (bill is-not george)
        (tom is-not george)
        (tom is-not bill)

        (rule if
              (and ((var x) likes (var y))
                   ((var z) likes (var y))
                   ((var x) is-not (var z)))
              then
              ((var x) friend-of (var z)))
        ))


;;; backward chaining with zeke the zebra
;;;try  ((var x) zeke)
;;;try  (animal (var x))
;;;try  ((var x) (var y))

(setq *assertions*
      '(
        (zebra zeke)
        (rule if (zebra (var x))
              then
              (herbivore (var x)))
        (rule if (giraffe (var x))
              then
              (herbivore (var x)))
        (rule if (tiger (var x))
              then
              (carnivore (var x)))
        (rule if (leopard (var x))
              then
```

```
              (carnivore (var x)))
        (rule if (herbivore (var x))
              then
              (mammal (var x)))
        (rule if (carnivore (var x))
              then
              (mammal (var x)))
        (rule if (mammal (var x))
              then
              (animal (var x)))
        ))


;;;more backward chaining with zeke
;;; this rulebase is branchier in the backward direction
;;;try  ((var x) zeke)
;;;try  (animal (var x))
;;;try  ((var x) (var y))

(setq *assertions*
      '(
        (zebra zeke)
        (rule if (zebra (var x))
              then
              (striped (var x)))
        (rule if (zebra (var x))
              then
              (height (var x) 10))
        (rule if (zebra (var x))
              then
              (herbivore (var x)))
        (rule if (striped (var x))
              then
              (not (solid (var x))))
        (rule if (striped (var x))
              then
              (not (spotted (var x))))
        (rule if (height (var x) 10)
              then
              (smaller-than-a-house (var x)))
        (rule if (height (var x) 10)
              then
              (bigger-than-a-breadbox (var x)))
        (rule if (herbivore (var x))
              then
              (mammal (var x)))
        (rule if (herbivore (var x))
              then
              (eats-vegetables (var x)))
        (rule if (mammal (var x))
              then
              (warm (var x)))
        (rule if (mammal (var x))
              then
```

```
              (animal (var x)))
        ))


;;;animal rule set with two fact bases for unknown animals
;;;try  (fred is (var x))
;;;try  ((var x) is cheetah)
;;;try  (tony is (var x))
;;;try  ((var x) is tiger)
;;;try  ((var x) is mammal)
;;;try  ((var x) is (var y))
;;;try  ((var x) has (var y))
;;;try  ((var x) has (var y) (var z))   explain the results
;;;try  ((var x) (var y) (var z))

(setq *assertions*
      '(
        (fred has dark spots)
        (fred has tawny color)
        (fred eats meat)
        (fred has hair)
        (tony has hair)
        (tony has pointed teeth)
        (tony has black stripes)
        (tony has claws)
        (tony has forward eyes)
        (tony has tawny color)

        (rule if ((var x) has hair)
              then
              ((var x) is mammal))
        (rule if ((var x) gives milk)
              then
              ((var x) is mammal))
        (rule if ((var x) has feathers)
              then
              ((var x) is bird))
        (rule if (and ((var x) flies)
                      ((var x) lays eggs))
              then
              ((var x) is bird))
        (rule if ((var x) eats meat)
              then
              ((var x) is carnivore))
        (rule if (and ((var x) has pointed teeth)
                      ((var x) has claws)
                      ((var x) has forward eyes))
              then
              ((var x) is carnivore))
        (rule if (and ((var x) is mammal)
                      ((var x) has hoofs))
              then
              ((var x) is ungulate))
        (rule if (and ((var x) is mammal)
```

```
                ((var x) chews cud))
          then
          ((var x) is ungulate))
        (rule if (and ((var x) is mammal)
                ((var x) is carnivore)
                ((var x) has tawny color)
                ((var x) has dark spots))
          then
          ((var x) is cheetah))
        (rule if (and ((var x) is mammal)
                ((var x) is carnivore)
                ((var x) has tawny color)
                ((var x) has black stripes))
          then
          ((var x) is tiger))
        (rule if (and ((var x) is ungulate)
                ((var x) has long neck)
                ((var x) has long legs)
                ((var x) has dark spots))
          then
          ((var x) is giraffe))
        (rule if (and ((var x) is ungulate)
                ((var x) has black stripes))
          then
          ((var x) is zebra))
        (rule if (and ((var x) is bird)
                ((var x) does not fly)
                ((var x) has long neck)
                ((var x) has long legs)
                ((var x) is black and white))
          then
          ((var x) is ostrich))
        (rule if (and ((var x) is bird)
                ((var x) does not fly)
                ((var x) swims)
                ((var x) is black and white))
          then
          ((var x) is penguin))
        (rule if (and ((var x) is bird)
                ((var x) flys well))
          then
          ((var x) is albatross))
        )  )


;;;relatives
;;;try  (mother (var x) Bert)
;;;try  (parent (var x) Bert)
;;;try  (sibling Carl (var z))     expalin the results
;;;try  (cousin (var x) (var y))
;;;try  (ancestor Art George)      explain the results

(setq *assertions*
      '(
```

```
        (mother Abby Bert)
        (mother Abby Betty)
        (mother Betty Claire)
        (mother Betty Carl)
        (mother Julia Karen)
        (father Art Bert)
        (father Art Betty)
        (father Gary Henry)
        (father Gary Hank)
        (father Carl George)
        (rule if (mother (var x) (var y))
              then
              (parent (var x) (var y)))
        (rule if (father (var x) (var y))
              then
              (parent (var x) (var y)))
        (rule if (parent (var x) (var y))
              then
              (ancestor (var x) (var y)))
        (rule if (and (ancestor (var x) (var y))
                      (parent (var y) (var z)))
              then
              (ancestor (var x) (var z)))
        (rule if (and (parent (var x1) (var y1))
                      (parent (var x2) (var y2))
                      (sibling (var x1) (var x2)))
              then
              (cousin (var y1) (var y2)))
        (rule if (and (parent (var x) (var y1))
                      (parent (var x) (var y2)))
              then
              (sibling (var y1) (var y2)))
        ))


;;;doing addition
;;;try  ((var x) + 6 = 1 1)
;;;try  ((var x) + (var y) = 0 6)
;;;try  (4 + (var x) = 1 (var y))
;;;try  ((var x) (var y) 6 = (var z) 0)
;;;try  (4 5 + 5 9 = (var x)(var y)(var z))
;;;try  (4 5 + (var x)(var y) = 1 0 4)
;;;try  ((var x) 7 + 5 (var y) = 0 (var z) 1)
;;;try  ((var x) (var y) + 5 (var z) = 0 (var y) (var x))
;;;try  ((var x) (var y) + 5 (var z) = (var x) (var y) (var x))

(setq *assertions*
      '(
        (1 + 0 = 0 1)
        (0 + 0 = 0 0)
        (0 + 1 = 0 1)
        (0 + 2 = 0 2)
        (0 + 3 = 0 3)
        (0 + 4 = 0 4)
```

(0 + 5 = 0 5)
(0 + 6 = 0 6)
(0 + 7 = 0 7)
(0 + 8 = 0 8)
(0 + 9 = 0 9)
(1 + 1 = 0 2)
(1 + 2 = 0 3)
(1 + 3 = 0 4)
(1 + 4 = 0 5)
(1 + 5 = 0 6)
(1 + 6 = 0 7)
(1 + 7 = 0 8)
(1 + 8 = 0 9)
(1 + 9 = 1 0)
(2 + 0 = 0 2)
(2 + 1 = 0 3)
(2 + 2 = 0 4)
(2 + 3 = 0 5)
(2 + 4 = 0 6)
(2 + 5 = 0 7)
(2 + 6 = 0 8)
(2 + 7 = 0 9)
(2 + 8 = 1 0)
(2 + 9 = 1 1)
(3 + 0 = 0 3)
(3 + 1 = 0 4)
(3 + 2 = 0 5)
(3 + 3 = 0 6)
(3 + 4 = 0 7)
(3 + 5 = 0 8)
(3 + 6 = 0 9)
(3 + 7 = 1 0)
(3 + 8 = 1 1)
(3 + 9 = 1 2)
(4 + 0 = 0 4)
(4 + 1 = 0 5)
(4 + 2 = 0 6)
(4 + 3 = 0 7)
(4 + 4 = 0 8)
(4 + 5 = 0 9)
(4 + 6 = 1 0)
(4 + 7 = 1 1)
(4 + 8 = 1 2)
(4 + 9 = 1 3)
(5 + 0 = 0 5)
(5 + 1 = 0 6)
(5 + 2 = 0 7)
(5 + 3 = 0 8)
(5 + 4 = 0 9)
(5 + 5 = 1 0)
(5 + 6 = 1 1)
(5 + 7 = 1 2)
(5 + 8 = 1 3)
(5 + 9 = 1 4)

```
        (6 + 0 = 0 6)
        (6 + 1 = 0 7)
        (6 + 2 = 0 8)
        (6 + 3 = 0 9)
        (6 + 4 = 1 0)
        (6 + 5 = 1 1)
        (6 + 6 = 1 2)
        (6 + 7 = 1 3)
        (6 + 8 = 1 4)
        (6 + 9 = 1 5)
        (7 + 0 = 0 7)
        (7 + 1 = 0 8)
        (7 + 2 = 0 9)
        (7 + 3 = 1 0)
        (7 + 4 = 1 1)
        (7 + 5 = 1 2)
        (7 + 6 = 1 3)
        (7 + 7 = 1 4)
        (7 + 8 = 1 5)
        (7 + 9 = 1 6)
        (8 + 0 = 0 8)
        (8 + 1 = 0 9)
        (8 + 2 = 1 0)
        (8 + 3 = 1 1)
        (8 + 4 = 1 2)
        (8 + 5 = 1 3)
        (8 + 6 = 1 4)
        (8 + 7 = 1 5)
        (8 + 8 = 1 6)
        (8 + 9 = 1 7)
        (9 + 0 = 0 9)
        (9 + 1 = 1 0)
        (9 + 2 = 1 1)
        (9 + 3 = 1 2)
        (9 + 4 = 1 3)
        (9 + 5 = 1 4)
        (9 + 6 = 1 5)
        (9 + 7 = 1 6)
        (9 + 8 = 1 7)
        (9 + 9 = 1 8)

        (rule if
              (and ((var b) + (var d) = (var x0) (var g))
                   ((var a) + (var c) = (var x1) (var x2))
                   ((var x0) + (var x2) = (var x3) (var f))
                   ((var x1) + (var x3) = 0 (var e)))
                   then
              ((var a) (var b) + (var c) (var d) = (var e) (var f) (var g)))
        ))

;;;built-in carry allows easier rules for adding large numbers
;;;try  (4 5 6 7 8 + 8 7 6 5 4 = (var a)(var b)(var c)(var d)(var e)(var f))
;;;try  (4 (var a) 6 + (var a) 3 (var b) = (var c) 6 (var b) (var a))
;;;try  ((var a)(var a) + (var b)(var b) = (var c)(var d)(var c))
```

```
;;;try  ((var a)(var a) + (var b)(var b) = (var c)(var b)(var c))
;;;don't try  ((var s)(var e)(var n)(var d) + (var m)(var o)(var r)(var e)
;;;            = (var m)(var o)(var n)(var e)(var y))

(setq *assertions*
      '(
        (0 + 0 + 0 = 0 0)
        (0 + 0 + 1 = 0 1)
        (0 + 0 + 2 = 0 2)
        (0 + 0 + 3 = 0 3)
        (0 + 0 + 4 = 0 4)
        (0 + 0 + 5 = 0 5)
        (0 + 0 + 6 = 0 6)
        (0 + 0 + 7 = 0 7)
        (0 + 0 + 8 = 0 8)
        (0 + 0 + 9 = 0 9)
        (0 + 1 + 0 = 0 1)
        (0 + 1 + 1 = 0 2)
        (0 + 1 + 2 = 0 3)
        (0 + 1 + 3 = 0 4)
        (0 + 1 + 4 = 0 5)
        (0 + 1 + 5 = 0 6)
        (0 + 1 + 6 = 0 7)
        (0 + 1 + 7 = 0 8)
        (0 + 1 + 8 = 0 9)
        (0 + 1 + 9 = 1 0)
        (0 + 2 + 0 = 0 2)
        (0 + 2 + 1 = 0 3)
        (0 + 2 + 2 = 0 4)
        (0 + 2 + 3 = 0 5)
        (0 + 2 + 4 = 0 6)
        (0 + 2 + 5 = 0 7)
        (0 + 2 + 6 = 0 8)
        (0 + 2 + 7 = 0 9)
        (0 + 2 + 8 = 1 0)
        (0 + 2 + 9 = 1 1)
        (0 + 3 + 0 = 0 3)
        (0 + 3 + 1 = 0 4)
        (0 + 3 + 2 = 0 5)
        (0 + 3 + 3 = 0 6)
        (0 + 3 + 4 = 0 7)
        (0 + 3 + 5 = 0 8)
        (0 + 3 + 6 = 0 9)
        (0 + 3 + 7 = 1 0)
        (0 + 3 + 8 = 1 1)
        (0 + 3 + 9 = 1 2)
        (0 + 4 + 0 = 0 4)
        (0 + 4 + 1 = 0 5)
        (0 + 4 + 2 = 0 6)
        (0 + 4 + 3 = 0 7)
        (0 + 4 + 4 = 0 8)
        (0 + 4 + 5 = 0 9)
        (0 + 4 + 6 = 1 0)
        (0 + 4 + 7 = 1 1)
```

(0 + 4 + 8 = 1 2)
(0 + 4 + 9 = 1 3)
(0 + 5 + 0 = 0 5)
(0 + 5 + 1 = 0 6)
(0 + 5 + 2 = 0 7)
(0 + 5 + 3 = 0 8)
(0 + 5 + 4 = 0 9)
(0 + 5 + 5 = 1 0)
(0 + 5 + 6 = 1 1)
(0 + 5 + 7 = 1 2)
(0 + 5 + 8 = 1 3)
(0 + 5 + 9 = 1 4)
(0 + 6 + 0 = 0 6)
(0 + 6 + 1 = 0 7)
(0 + 6 + 2 = 0 8)
(0 + 6 + 3 = 0 9)
(0 + 6 + 4 = 1 0)
(0 + 6 + 5 = 1 1)
(0 + 6 + 6 = 1 2)
(0 + 6 + 7 = 1 3)
(0 + 6 + 8 = 1 4)
(0 + 6 + 9 = 1 5)
(0 + 7 + 0 = 0 7)
(0 + 7 + 1 = 0 8)
(0 + 7 + 2 = 0 9)
(0 + 7 + 3 = 1 0)
(0 + 7 + 4 = 1 1)
(0 + 7 + 5 = 1 2)
(0 + 7 + 6 = 1 3)
(0 + 7 + 7 = 1 4)
(0 + 7 + 8 = 1 5)
(0 + 7 + 9 = 1 6)
(0 + 8 + 0 = 0 8)
(0 + 8 + 1 = 0 9)
(0 + 8 + 2 = 1 0)
(0 + 8 + 3 = 1 1)
(0 + 8 + 4 = 1 2)
(0 + 8 + 5 = 1 3)
(0 + 8 + 6 = 1 4)
(0 + 8 + 7 = 1 5)
(0 + 8 + 8 = 1 6)
(0 + 8 + 9 = 1 7)
(0 + 9 + 0 = 0 9)
(0 + 9 + 1 = 1 0)
(0 + 9 + 2 = 1 1)
(0 + 9 + 3 = 1 2)
(0 + 9 + 4 = 1 3)
(0 + 9 + 5 = 1 4)
(0 + 9 + 6 = 1 5)
(0 + 9 + 7 = 1 6)
(0 + 9 + 8 = 1 7)
(0 + 9 + 9 = 1 8)

(1 + 0 + 0 = 0 1)

(1 + 0 + 1 = 0 2)
(1 + 0 + 2 = 0 3)
(1 + 0 + 3 = 0 4)
(1 + 0 + 4 = 0 5)
(1 + 0 + 5 = 0 6)
(1 + 0 + 6 = 0 7)
(1 + 0 + 7 = 0 8)
(1 + 0 + 8 = 0 9)
(1 + 0 + 9 = 1 0)
(1 + 1 + 0 = 0 2)
(1 + 1 + 1 = 0 3)
(1 + 1 + 2 = 0 4)
(1 + 1 + 3 = 0 5)
(1 + 1 + 4 = 0 6)
(1 + 1 + 5 = 0 7)
(1 + 1 + 6 = 0 8)
(1 + 1 + 7 = 0 9)
(1 + 1 + 8 = 1 0)
(1 + 1 + 9 = 1 1)
(1 + 2 + 0 = 0 3)
(1 + 2 + 1 = 0 4)
(1 + 2 + 2 = 0 5)
(1 + 2 + 3 = 0 6)
(1 + 2 + 4 = 0 7)
(1 + 2 + 5 = 0 8)
(1 + 2 + 6 = 0 9)
(1 + 2 + 7 = 1 0)
(1 + 2 + 8 = 1 1)
(1 + 2 + 9 = 1 2)
(1 + 3 + 0 = 0 4)
(1 + 3 + 1 = 0 5)
(1 + 3 + 2 = 0 6)
(1 + 3 + 3 = 0 7)
(1 + 3 + 4 = 0 8)
(1 + 3 + 5 = 0 9)
(1 + 3 + 6 = 1 0)
(1 + 3 + 7 = 1 1)
(1 + 3 + 8 = 1 2)
(1 + 3 + 9 = 1 3)
(1 + 4 + 0 = 0 5)
(1 + 4 + 1 = 0 6)
(1 + 4 + 2 = 0 7)
(1 + 4 + 3 = 0 8)
(1 + 4 + 4 = 0 9)
(1 + 4 + 5 = 1 0)
(1 + 4 + 6 = 1 1)
(1 + 4 + 7 = 1 2)
(1 + 4 + 8 = 1 3)
(1 + 4 + 9 = 1 4)
(1 + 5 + 0 = 0 6)
(1 + 5 + 1 = 0 7)
(1 + 5 + 2 = 0 8)
(1 + 5 + 3 = 0 9)
(1 + 5 + 4 = 1 0)

```
(1 + 5 + 5 = 1 1)
(1 + 5 + 6 = 1 2)
(1 + 5 + 7 = 1 3)
(1 + 5 + 8 = 1 4)
(1 + 5 + 9 = 1 5)
(1 + 6 + 0 = 0 7)
(1 + 6 + 1 = 0 8)
(1 + 6 + 2 = 0 9)
(1 + 6 + 3 = 1 0)
(1 + 6 + 4 = 1 1)
(1 + 6 + 5 = 1 2)
(1 + 6 + 6 = 1 3)
(1 + 6 + 7 = 1 4)
(1 + 6 + 8 = 1 5)
(1 + 6 + 9 = 1 6)
(1 + 7 + 0 = 0 8)
(1 + 7 + 1 = 0 9)
(1 + 7 + 2 = 1 0)
(1 + 7 + 3 = 1 1)
(1 + 7 + 4 = 1 2)
(1 + 7 + 5 = 1 3)
(1 + 7 + 6 = 1 4)
(1 + 7 + 7 = 1 5)
(1 + 7 + 8 = 1 6)
(1 + 7 + 9 = 1 7)
(1 + 8 + 0 = 0 9)
(1 + 8 + 1 = 1 0)
(1 + 8 + 2 = 1 1)
(1 + 8 + 3 = 1 2)
(1 + 8 + 4 = 1 3)
(1 + 8 + 5 = 1 4)
(1 + 8 + 6 = 1 5)
(1 + 8 + 7 = 1 6)
(1 + 8 + 8 = 1 7)
(1 + 8 + 9 = 1 8)
(1 + 9 + 0 = 1 0)
(1 + 9 + 1 = 1 1)
(1 + 9 + 2 = 1 2)
(1 + 9 + 3 = 1 3)
(1 + 9 + 4 = 1 4)
(1 + 9 + 5 = 1 5)
(1 + 9 + 6 = 1 6)
(1 + 9 + 7 = 1 7)
(1 + 9 + 8 = 1 8)
(1 + 9 + 9 = 1 9)

(rule if
      (and (       0 + (var a0) + (var b0) = (var c1) (var r0))
           ((var c1) + (var a1) + (var b1) = (var c2) (var r1)))
           then
      (           (var a1) (var a0)
                +  (var b1) (var b0)
       = (var c2) (var r1) (var r0)))
```

```
        (rule if
              (and (       0 + (var a0) + (var b0) = (var c1) (var r0))
                   ((var c1) + (var a1) + (var b1) = (var c2) (var r1))
                   ((var c2) + (var a2) + (var b2) = (var c3) (var r2)))
                   then
              (           (var a2) (var a1) (var a0)
                        +  (var b2) (var b1) (var b0)
               = (var c3) (var r2) (var r1) (var r0)))

        (rule if
              (and (       0 + (var a0) + (var b0) = (var c1) (var r0))
                   ((var c1) + (var a1) + (var b1) = (var c2) (var r1))
                   ((var c2) + (var a2) + (var b2) = (var c3) (var r2))
                   ((var c3) + (var a3) + (var b3) = (var c4) (var r3)))
                   then
              (           (var a3) (var a2) (var a1) (var a0)
                        +  (var b3) (var b2) (var b1) (var b0)
               = (var c4) (var r3) (var r2) (var r1) (var r0)))

        (rule if
              (and (       0 + (var a0) + (var b0) = (var c1) (var r0))
                   ((var c1) + (var a1) + (var b1) = (var c2) (var r1))
                   ((var c2) + (var a2) + (var b2) = (var c3) (var r2))
                   ((var c3) + (var a3) + (var b3) = (var c4) (var r3))
                   ((var c4) + (var a4) + (var b4) = (var c5) (var r4)))
                   then
              (           (var a4) (var a3) (var a2) (var a1) (var a0)
                        +  (var b4) (var b3) (var b2) (var b1) (var b0)
               = (var c5) (var r4) (var r3) (var r2) (var r1) (var r0)))
        ))

|#
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;OBJECT-ORIENTED PROGRAMMING
;;;
;;;  code from George Lugar and William Stubblefield
;;;    Artificial Intelligence and the Design of Expert Systems
;;;    Benjamin/Cummings, 1989

;;;define a new object, given its parent and an optional list
;;; of instance variables and bindings
(defun def-object (obj parent &optional vars)
  (setf (get obj 'ISA) parent)
  (setf (get obj 'VARIABLES) (evaluate-bindings vars)))

(defun evaluate-bindings (vars)
  (mapcar #'bind-eval-fn vars))

(defun bind-eval-fn (x)
  (list (first x) (eval (second x))))

;;;attach methods to an object
(defun def-method (obj name def)
  (setf (get obj 'METHODS)
        (replace-method name def (get obj 'METHODS))))

(defun replace-method (name def method-lst)
  (cond ((null method-lst)
         (acons name def nil))
        ((equal name (first (first method-lst)))
         (acons name def (rest method-lst)))
        (T (cons (first method-lst)
                 (replace-method name def (rest method-lst))))))

;;;search inheritance hierarchy
(defun inherit-method (obj method)
  (cond ((null obj) nil)
        ((atom obj)
         (or (get-method obj method)
             (inherit-method (get obj 'ISA) method)))
        (T (or (inherit-method (first obj) method)
               (inherit-method (rest obj) method)))))

(defun get-method (obj name)
  (cdr (assoc name (get obj 'METHODS))))

;;;evaluate a message to an object
(defun send (obj method &rest method-parameters)
  (let ((meth (inherit-method obj method))
        env)
    (cond (meth
           (setq env (build-env obj))
           (progv (cons 'SELF (mapcar #'first env))
                  (cons obj (mapcar #'second env))
             (apply meth method-parameters)))
```

```
          (T (print (list 'Unknown 'method method))))))

;;;walks the inheritance hierarchy, depth-first, constructing
;;; a list of inherited variable-binding pairs.
(defun build-env (obj)
  (cond ((null obj) nil)
        ((listp obj)
         (append (build-env (rest obj)) (build-env (first obj))))
        (T (append (build-env (get obj 'ISA))
                   (get obj 'VARIABLES)))))

(defun build-root ()
  (def-object 'ROOT nil)
  (def-method 'ROOT 'SHOW
    #'(lambda ()
        (terpri)
        (print (list self 'has 'parents))
        (pprint (get self 'ISA))
        (terpri)
        (print (list self 'has 'attached 'variables))
        (pprint (get self 'VARIABLES))
        (terpri)
        (print (list self 'has 'attached 'methods))
        (pprint (get self 'METHODS))
        (terpri)))
  (def-method 'ROOT 'SHOW-PARENTS
    #'(lambda ()
        (get self 'ISA)))
  (def-method 'ROOT 'SHOW-VALUE
    #'(lambda (name)
        (eval name)))
  (def-method 'ROOT 'SHOW-ENV
    #'(lambda ()
        (build-env self)))
  (def-method 'ROOT 'SET-VALUE
    #'(lambda (var value)
        (let ((pair (assoc var (get self 'VARIABLES))))
          (cond (pair (rplacd pair (list value)))
                (T (setf (get self 'VARIABLES)
                         (cons (list var value)
                               (get self 'VARIABLES))))))))
  '*ROOT-WORLD-READY*)


;;;;;;;;;;;;;;;;;;;;;;;;;
;;;object worlds
;;;rectangles and squares

(defun make-rectangle-world ()
  (def-object 'rectangle 'root
    '((numsides 4)
      (description "Four-sided planar figure, all angles = 90
degrees")))
  (def-object 'rectangle-1 'rectangle
```

```
    '((length 8)
      (width 4)))
  (def-object 'square 'rectangle
    '((description "Rectangle with equal sides")))
  (def-object 'square-1 'square
    '((side 10)))
  (def-method 'rectangle 'area
    #'(lambda ()
       (* length width)))
  (def-method 'square 'area
    #'(lambda ()
       (* side side)))
  '*RECTANGLE-WORLD-READY*)

;;;;;;;;;;;;;;;;;;;;;;;
;;;object-world
;;;rooms and thermostats

(defun make-thermostat-world ()
  (def-object 'thermostat 'root
    '((setting 65)))
  (def-object 'room 'root
    '((temperature 65)))
  (def-object 'heater 'root
    '((state 'off)))
  (def-object 'room-311 'room
    '((thermostat 'thermostat-311)))
  (def-object 'thermostat-311 'thermostat
    '((heater 'heater-311)
      (location 'room-311)))
  (def-object 'heater-311 'heater
    '((location 'room-311)))
  (def-method 'room 'change-temp
    #'(lambda (amount-of-change)
       (let ((new-temp (+ amount-of-change temperature)))
         (send self 'set-value 'temperature new-temp)
         (print
          (list 'Temperature 'in self 'changes 'to new-temp 'degrees.))
         (send thermostat 'check-temp))))
  (def-method 'thermostat 'check-temp
    #'(lambda ()
       (cond ((< (send location 'show-value 'temperature) setting)
              (send heater 'turn-on))
             (T (send heater 'turn-off)))))
  (def-method 'thermostat 'change-setting
    #'(lambda (temp)
       (send self 'set-value 'setting temp)
       (print (list 'New 'setting 'of self 'is temp 'degrees.))
       (send self 'check-temp)))
  (def-method 'heater 'turn-on
    #'(lambda ()
       (cond ()
             ((equal state 'off)
              (print (list 'Heater 'turns 'on 'in location))
```

```
              (send self 'set-value 'state 'on)))
        (send location 'change-temp 1)))
  (def-method 'heater 'turn-off
    #'(lambda ()
        (cond ()
              ((equal state 'on)
               (print (list 'Heater 'turns 'off 'in location))
               (send self 'set-value 'state 'off)))))
  '*THERMOSTAT-WORLD-READY*)

#|
;;;try this after loading the object file
;;; ignore the compiler warnings, this implementation is not smart
;;; about the dynamic binding of object values.  If it were smart,
;;; the code would look a lot more like the stream code.

(build-root)
(pprint (symbol-plist 'root))

;;;note that the methods functions are not visible, another weakness
;;; of the naive implementation
;;; here's the method-call for showing the symbol property list:

(send 'root 'show)

(make-rectangle-world)
(send 'rectangle 'show-value 'description)
(send 'rectangle-1 'show-value 'width)
(send 'rectangle-1 'show-value 'length)
(send 'rectangle-1 'area)
(send 'rectangle-1 'set-value 'width 6)
(send 'rectangle-1 'area)
(send 'square-1 'show-parents)
(send 'square 'show-parents)
(send 'rectangle 'show-parents)
(send 'square-1 'show-env)
(send 'square-1 'area)
(send 'square-1 'set-value 'side 14)
(send 'square-1 'area)

(make-thermostat-world)
(send 'room-311 'show-value 'temperature)
(send 'thermostat-311 'show-value 'setting)
(send 'heater-311 'show-value 'state)
(send 'room-311 'change-temp -5)
(send 'thermostat-311 'change-setting 70)
|#
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;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))))))

|#
```