```
;; CA1.LSP -- contains most of the control structure for revised version of CA.
;; (rewritten in LISP by M. Burstein 11/7/79 from CA1.MLL)


;; CA  is the top-level function.
(de CA nil
  (PROG nil

;; :changed-cons holds concepts changed or added to :C-list during one pass of CA
        (setq :changed-cons nil)
        (setq :current-phrase nil)
        (setq !current-DOUBLE #CA)
;;If the current sentence is finished, then get a new one and re-initialize;
        (cond ((NULL :sent) (init-ca)))
        (cond ((NULL :sent) (setq !WORKING nil) (return nil)))
        (pms; #CA1OP1 T "Entering CA:" T "  :C-list  = " (delinfree))
;;Set :word and :next-word, and look them up in the dictionary;
 CYCLE: (get-next-item)
        (cond ((NULL :word) (return nil)))
        (pms; #CA T "Beginning main execution cycle;"
                  T ":word = " :word
                  T ":sent = " :sent
                  T ":C-list = " (delinfree)
                  T ":request-pools = " :request-pools)
;;Remove request pools which no longer contain active requests
        (clean-up-request-pools)
        (consider-lexical-requests)
;;If next word ends noun group, then return to regular mode immediately,
;;so that next call to consider-requests (below) will consider all requests;
        (cond ((and (flagon #noun-group-flag) (end-noun-phrase))
               (remove-flag #noun-group-flag)
               (add-flag #results-flag)
               (pms; #CA1 T "End of noun group;")))
;;Activate requests associated with :word;
        (activate-item-requests)
        (consider-requests)

;;If next word begins a noun group, then begin noun-group mode immediately,
;;so that the next call to consider-requests (above, but later in the loop),
;;will consider only latest requests;
        (cond ((and (not (flagon #noun-group-flag))
                    (setq :N-P-record (begin-noun-phrase)))
               (add-flag #noun-group-flag)
               (cond (:changed-cons
                      (pms; #CA1 T "Begin noun group;")))))

;;If any add-con has built a CO which has a lot of memory stuff hooked onto
;;it, or if the next word might mark the end of the clause, then return to
;;let memory do its thing;
        (cond ((or (flagon #MEMORY-HOOK-flag)
                   (CLAUSE-POINT :next-word)
                   (flagon #results-flag))
               (remove-flag #MEMORY-HOOK-flag)
               (remove-flag #results-flag)
               (pms; #CA1OP T "CA processed words: " :current-phrase
                     T "  :C-list = " (delinfree)
                     T)
               (setq :changed-cons (REPL-UNSET-cons :changed-cons))
               (setq !current-DOUBLE nil)
               (return T)))
```

```
            (T (GO CYCLE:)))))

~This initializes some variables
(de init-ca-vars nil
~These three are just book-keeping globals containing generated genSYM's
 (setq :ALL-reqs nil)
 (setq :ALL-cons nil)
 (setq :ALL-pools nil)
 (setq :all-lexes nil)
 (setq :word nil)                    ~:word is the current lexical item;
 (setq :next-word nil)               ~:next-word is the next lexical item;
 (setq :C-list nil)        ~:C-list is the concept list, ie. parsing STM
 (setq :request-pools nil)           ~ the list of request pools;
 (setq :extra-requests nil)  ~holds reqs to be activated in next pool
                             ~ (this enables dynamic word sense changes)
 (setq :flagS nil)                   ~:flagS holds all the flags;
 (setq :N-P-record nil)              ~a global used by end-noun-phrase
 (setq :LAST-EMBEDded-Con nil)       ~holds the last thing embedded
 (setq :changed-cons nil))



~This initializes the program;
(de init-ca nil
 (init-ca-vars)
 (setq :sentENCE (next-sentENCE)) ~Get the next sentence of the story;
 (setq :sent :sentENCE)           ~:sent is just a copy of sentENCE;
 (pmsg @INITCA T "init-ca new sentence is " T :sent T)
 (putprop @:lexical-pool nil @requests)  ~:lexical-pool holds requests
                                         ~  looking for specific words.
~  initialize :next-word
 (cond ((EQUAL (setq :next-word (car :sent)) @*)
        (add-flag @CHANGE-TRACE-flag)
        (setq :sent (cdr :sent))
        (setq :next-word (car :sent))))
 (get-lex-into '!next-lex :next-word)     ~look up the definition
 (cond ((setq :N-P-record (begin-noun-phrase))
        (add-flag @noun-group-flag)
        (pmsg @CAT T "begin noun group;")))

(de next-sentENCE () (POP !INPUT))

(de list-add (C L)
     (or (MEMQ C (eval L)) (SET L (cons C (eval L)))))


~This sets :word, :next-word, and :sent, as well as looking up :next-word;
~the first thing, though, is to check if the * flag was set; if so, change
~the pause flag;
(de get-next-item ()
        (cond ((flagon @CHANGE-TRACE-flag)
               (setq :PAUSE (not :PAUSE))
               (remove-flag @CHANGE-TRACE-flag)))
        (setq :word :next-word)
        (and :word (SET :word :word))
        (setq :sent (cdr :sent))
        (cond ((EQUAL (setq :next-word (car :sent)) @*)
               (add-flag @CHANGE-TRACE-flag)
               (setq :sent (cdr :sent))
               (setq :next-word (car :sent)))))
```

```
        (setq !lex !next-lex)
        (get-lex-info '!next-lex :next-word)
        (pmsg @CAT T "The current word is " :word ";")
        (setq :current-phrase (nconc1 :current-phrase :word))
        :word)

;Put-first takes an inactivated pool and a request and puts the request at the
; beginning of the pool.  If the request already exists, it is removed from
;its previous position
(de put-first (req pool)
 (cond ((memq req (get pool 'specs:))
        (putprop pool (remove req (get pool 'specs:)) 'specs:)))
 (putprop pool (cons req (get pool 'specs:)) 'specs:))


; Removes pools which no longer contain active requests from :request-pools
(de clean-up-request-pools ()
  (clean-up-special-pools)
  (setq :request-pools
   (mapcan (funl (P)
             (and (live-pool P) (ncons P)))
           :request-pools)))


; pool still contains some live requests.
(de live-pool (pool)
          (SOME (funl (R) (get R @active))
                (get pool @requests)))


; Returns list of live requests in pool
(de SAVE-live-reqs (pool)
        (putprop pool
                 (mapcan (funl (R) (and (get R @active) (ncons R)))
                         (get pool @requests))
                 @requests))

(de clean-up-special-pools ()
      (SAVE-live-reqs @:lexical-pool)
      )



;This routine considers the requests with tests which are looking for
;specific words, which are contained in :LEXICAL-pool
(de consider-lexical-requests ()
    (cond ((get @:lexical-pool @requests)
           (pmsg @CAT T "Considering lexical requests;")
           (consider-pool @:lexical-pool))))


;Considers requests depending on mode setting
(de consider-requests ()
  (cond ((flagon @noun-group-flag)
         (pmsg @CAT T "Considering latest requests;")
         (consider-LATEST-requests))
        (T (pmsg @CAT T "Considering all requests;")
           (consider-ALL-requests))))
```

```
;This eventually needs to be more sophisticated, handling the problem of
;request priority within a given pool; it returns nil unless at least one
;of the calls to consider returned T;
(de consider-pool (pool)
  (mapcan (function (req)
              (and (consider req) (ncons T)))
          (get pool 'requests)))



%
This is a bit changed from the original description; in order to be somewhat
more "depth first", whenever a request is triggered, instead of continuing
down the list of request pools, the program immediately goes back to the most
recent ones again; this is especially important if any new requests were
spawned; note that at present, consider-pool does not operate that way, so
all the requests in a pool get considered even if one has already triggered;
(obviously, :request-pools must be ordered with more recent pools appearing
after later ones in the list);
:request-pools no longer REVERSED every time through. - Kept as stack. mp
%
(de consider-ALL-requests ()
   (DO WHILE (SOME 'consider-pool :request-pools)))


;This only considers requests in the most recent pool, including any
;requests spawned by them; (comments above apply here too);
(de consider-LATEST-requests ()
  (let (LATEST-pool (car :request-pools))
      (DO WHILE (SOME 'consider-pool
                      ; list of pools from LATEST-pool to front of list.
                      (LDIFF :request-pools
                             (cdr (MEMBER LATEST-pool :request-pools)))))))


;Checks each test-action pair of a request in turn; if some test true,
;; then associated action is evaled.
;; Returns T iff the request was triggered.
(de consider (request)
  (let (!new-C>1 nil)
   (cond ((get request 'active)
          (SOME (function (CLAUSE)
                  (cond ((eval-TEST CLAUSE)
                         (prog2 'CA T request " has fired;" T ":C-list = "
                                (delinfree) T)
                         (putprop request nil 'active)
                         (eval-ACTION CLAUSE)
                          )))
                (get request 'body))
;                       if  request fired (active prop is nil) then return T
;                       also, if NO-KILL-flag set, then reactivate the request.
          (cond ((NULL (get request 'active))
                 (cond ((flagon 'NO-KILL-flag)
                        (remove-flag 'NO-KILL-flag)
                        (putprop request T 'active)))
                 T)
                (T nil))))))


;; Evaluate the text of a request clause.
```

```
;Use of ::= in test is done by a fexpr.
(de eval-TEST (CL) (eval (CADAR CL)))


;
::= handles pseudo-assignments within the tests of requests; it works by
changing the value of CLAUSE, which is local to consider but free here;
changing CLAUSE appropriately communicates the effect of a pseudo-assignment
to eval-ACTION; N.B. ::= the function only works when called from eval-TEST;
the appearance of this symbol for pseudo-assignment in the actions of a
request looks the same, but in that case the work is done by eval-ACTION;
;
(DFX ::= (X)
  (let (BINDING (eval (cadr X)))
       (cond (BINDING
              (setq CLAUSE (SUBST (list 'QUOTE BINDING) (car X) CLAUSE))
              T)
             (T nil))))

;Evaluate the actions in a request clause
;Note that if a pseudo-assignment appears in an action, eval-ACTION itself
;does the work; the function ::= is not called;
(de eval-ACTION (CL)
  (PROG (ACTION ACT-list)
        (setq ACT-list (CDADR CL))
  LOOP: (cond ((NULL ACT-list) (return T)))
        (setq ACTION (car ACT-list))
        (setq ACT-list (cdr ACT-list))
        (cond ((EQUAL (car ACTION) '::=)
               (setq ACT-list
                     (SUBST (list 'QUOTE (eval (CADDR ACTION)))
                            (cadr ACTION)
                            ACT-list)))
              (T (eval ACTION)))
        (GO LOOP:)))


;Activates requests associated with the current word, if any.
;:extra-requests holds requests built by requests in :lexical-pool
;for activation in the next request pool built, which is here;
(de activate-item-requests ()
  (cond ((or :extra-requests (get !lex 'pool:) 'specs:)
         (activate-pool
          (append :extra-requests
                  (cond ((flagon 'skip-word-flag)
                         (remove-flag 'skip-word-flag)
                         nil)
                        (T (make-requests (get (get !lex 'pool:)'requests:))))))
         (setq :extra-requests nil))))


;When a request knows what to do with the next word (by checking get-next-item)
;  then don't load the definitions under the new word.
;     (see activate-item-requests)
(de skip-next-word () (add-flag 'skip-word-flag))

;Build-pool takes a dictionary and builds it into an active stucture;
;that means constructing a request pool out of it.
(de build-pool (new-requests)
  (let (pool (new-pool))
```

```
        (putprop pool new-requests #specs:)
        pool))

%
Activate-pool takes a pool which has been built using Build-pool
and puts in on the list of request pools.
%
(de activate-pool (pool)
 (setq :request-pools (cons pool :request-pools)))


~Given a list of requests in dictionary format, this puts them into
~active format.
(de make-requests (reqs)
  (mapcar (function gen-request) reqs))


~Changes one request from dictionary format to active format
(de gen-request (R)
  (let (request (new-req))
        (cond ((atom R) (setq R (eval R))))   ~for named requests.
        (setq R (cdr R))          ~removes word "request from front of request.
        (putprop request (clausify R) #body)
        (putprop request T #active)
        request))


~clausify filters out non-request-related information from a lexical
~specification
(de clausify (x) (split-clauses (for (y in x) (filter
                                                (and (neq (car y) '(test action)
                                                 y))))))

~Changes a list of test-action pairs from dictionary format to internal
~clause format.    (T A T A ...) => ((T A) (T A) ...)
(de split-clauses (X)
  (and X (append (list (list (car X) (cadr X)))
                 (split-clauses (cddr X)))))
```