*File created:* 17-Apr-86 16:28:32 **{PHYLUM}<STANSBURY>PARSER>RELEASE.1>PARSER.;10**

*changes to:* (FNS PG.BACKUP PG.RESOLVE PG.BUILD.LOOKAHEAD.SETS PG.CODE.PARSER)

*previous date:* 4-Apr-86 16:29:22 {PHYLUM}<STANSBURY>PARSER>RELEASE.1>PARSER.;7

*Read Table:* OLD-INTERLISP-FILE

*Package:* INTERLISP

*Format:* XCCS

```
(* * Copyright (c) 1983, 1984, 1986 by Xerox Corporation. All rights reserved.)

(RPAQQ PARSERCOMS
       ((* * Parser generation system.)
        (FNS MAKEPARSER PG.INITIALIZE.GRAMMAR)
        (RECORDS PARSERSPEC GRAMMAR)
        (DECLARE: EVAL@COMPILE DONTCOPY (RECORDS FPRODUCTION ALTERNATIVE))
        (MACROS SELF STRICTEOF TCONC.FRONT)
        (* * State machine generation. For an explanation of the parser technology, see "Theory & Construction
           of LR(k) Parsers" (Preliminary Version)
           , Benjamin M. Brosgol, Center for Research in Computing Technology, Harvard University, March 1973)
        (FNS PG.LR0FSM PG.AFSMS PG.ROOTPRODUCTION PG.AFSM.ADDPROD PG.CONNECT PG.ADD.ARC PG.DETERMINISTIC
             PG.DETERMINISTIC1 PG.CONNECT.AFSMS PG.CONNECT.AFSMS1 PG.SINGLETON PG.CONNECTED PG.OWNS.LINKS
             PG.RABIN.SCOTT PG.RABIN.SCOTT1 PG.REPLACEMENT PG.TRANSITION.SYMBOLS PG.NEXT.STATES
             PG.COMPOSITE.STATE.NAME PG.DISCONNECT PG.NONTERMINALP PG.TERMINALP)
        (FNS PG.LALRKFSM PG.BUILD.BackLinks PG.CONNECT.BACK PG.STATETYPE PG.RESOLVE PG.PRINT.INADEQUATE
             PG.BUILD.LOOKAHEAD.SETS PG.DISJOINT PG.LLA PG.LLA.LOOKAHEAD PG.LLA.READ PG.LLA.REDUCE PG.BACKUP
             PG.LOOKAHEAD.SOURCE PG.REDUCE.TARGET PG.LOOKAHEADP)
        (FNS PG.OLD.LLA)
        (DECLARE: EVAL@COMPILE DONTCOPY (RECORDS PRODUCTION FSM STATE LINK))
        (INITRECORDS PRODUCTION FSM STATE LINK)
        (FNS PG.SMASH.FSM)
        (FNS PG.PPL PG.PPM PG.PPP PG.PPS)
        (FNS PG.STATES PG.STATES1 PG.STATE.ORDER)
        (* * Lisp code generation)
        (FNS PG.CODE.PARSER PG.CODE.STATES)
        (FNS PG.CODE.LOOKAHEAD PG.CODE.LOOKAHEAD.ALL.TOKENS PG.CODE.LOOKAHEAD.TOKEN PG.CODE.LOOKAHEAD.SWITCH
             PG.CODE.LOOKAHEAD.MATCH)
        (FNS PG.CODE.READ PG.CODE.READ.TOKEN PG.CODE.READ.SWITCH)
        (FNS PG.CODE.REDUCE PG.CODE.REDUCE.STACKS PG.CODE.REDUCE.SWITCH)
        (* * Other)
        (FNS PRINT.FUNCTION)))
```

(* * Parser generation system.)

```
(DEFINEQ
```

## (MAKEPARSER

```
  [LAMBDA (PARSERSPEC SORTED?)                                    (* hts: " 3-Apr-86 16:20")
```

(* * Constructs a parser according to PARSERSPEC)

```
    (LET ((GRAMMAR (fetch GRAMMAR of PARSERSPEC))
          (NAME (fetch PARSERNAME of PARSERSPEC)))
         (PG.INITIALIZE.GRAMMAR GRAMMAR)
         (LET [(M (PG.LALRKFSM (PG.LR0FSM GRAMMAR]
              (/PUTD NAME (PG.CODE.PARSER M PARSERSPEC SORTED?))
              (PG.SMASH.FSM M))
        NAME])
```

## (PG.INITIALIZE.GRAMMAR

```
  [LAMBDA (G)                                                     (* hts: "28-Feb-86 17:45")
```

(* * Builds the SymbolTable for GRAMMAR G. The SymbolTable is a hashtable which maps each symbol in the alphabet of grammar G onto {NONTERMINAL, TERMINAL}. Also checks the grammar for syntactic correctness)

```
    (LET ((PRODS (fetch PRODUCTIONS of G))
          (TABLE (fetch (GRAMMAR SymbolTable) of G)))
```

(* * Note the nonterminals)

```
         (CLRHASH TABLE)
         (for P in PRODS do (if (NOT (type? FPRODUCTION P))
                                then (ERROR "Improper grammar format"))
                            (PUTHASH (fetch LEFTHAND of P)
                                   (QUOTE NONTERMINAL)
                                   TABLE))
```

(* * Note the terminals)

```
         [for P in PRODS do (for A in (fetch ALTERNATIVES of P)
                               do (if (NOT (type? ALTERNATIVE A))
                                      then (ERROR "Improper grammar format"))
```

```
                                (for SYMBOL in (fetch RULE of A) do (if (NOT (ATOM SYMBOL))
                                                                        then (ERROR "Improper grammar format"))
                                                                    (if (NULL (GETHASH SYMBOL TABLE))
                                                                        then (PUTHASH SYMBOL (QUOTE TERMINAL)
                                                                                     TABLE]

         (PUTHASH (QUOTE EOF)
                (QUOTE TERMINAL)
                TABLE)
         G])

)

(DECLARE: EVAL@COMPILE

(DATATYPE PARSERSPEC (PARSERNAME                          (* name to be given to parser)
                           GRAMMAR                        (* specifies language to be parsed)
                           READFN                         (* called to read next token)
                           CLASSFN                        (* called to determine class of a token)
                           INSTANCEFN                     (* called to determine instance of a token)
                           EOFFN                          (* called to verify that token read may be interpreted as EOF)
                           STACKINITFN                    (* initializes empty stacks)
                           PUSHFN                         (* push method for stacks)
                           POPFN                          (* pop method for stacks)
                           TOPFN                          (* tells what's on top of the stack)
                           LAQUEUEINITFN                  (* initializes main lookahead queue)
                           QUEUEINITFN                    (* initializes empty temp lookahead queue)
                           ENQUEUEFN                      (* enqueueing method for queues)
                           DEQUEUEFN                      (* dequeueing method for queues)
                           QUEUENOTEMPTYFN                (* tells if queue is empty)
                           SAVESTATEFN                    (* bundles up state to be saved)
                           )
       CLASSFN _ (QUOTE SELF)
       INSTANCEFN _ (QUOTE SELF)
       EOFFN _ (QUOTE STRICTEOF)
       STACKINITFN _ (QUOTE NILL)
       PUSHFN _ (QUOTE push)
       POPFN _ (QUOTE pop)
       TOPFN _ (QUOTE CAR)
       LAQUEUEINITFN _ (QUOTE SELF)
       QUEUEINITFN _ (QUOTE CONS)
       ENQUEUEFN _ (QUOTE TCONC)
       DEQUEUEFN _ (QUOTE TCONC.FRONT)
       QUEUENOTEMPTYFN _ (QUOTE CDR)
       SAVESTATEFN _ (QUOTE SELF))

(DATATYPE GRAMMAR (StartSymbol                            (* the atom which is the start symbol)
                         PRODUCTIONS                      (* rules; should be a list of FPRODUCTIONs)
                         SymbolTable                      (* hasharray which tells class of symbols)
                         )
       SymbolTable _ (HASHARRAY 33))
)

(/DECLAREDATATYPE (QUOTE PARSERSPEC)
       (QUOTE (POINTER POINTER POINTER POINTER POINTER POINTER POINTER POINTER POINTER POINTER POINTER POINTER
                  POINTER POINTER POINTER POINTER))
       ;; ---field descriptor list elided by lister---
       (QUOTE 32))

(/DECLAREDATATYPE (QUOTE GRAMMAR)
       (QUOTE (POINTER POINTER POINTER))
       ;; ---field descriptor list elided by lister---
       (QUOTE 6))

(DECLARE: EVAL@COMPILE DONTCOPY

(DECLARE: EVAL@COMPILE

[RECORD FPRODUCTION (LEFTHAND . ALTERNATIVES)

          (* * LEFTHAND is the left-hand side of the rule; ALTERNATIVES is a list of ALTERNATIVEs, which are alternative
          right-hand sides)

       (TYPE? (AND (LISTP DATUM)
                   (LITATOM (fetch LEFTHAND of DATUM))
                   (LISTP (fetch ALTERNATIVES of DATUM]

[RECORD ALTERNATIVE (RULE AUGMENT)
       (TYPE? (AND (LISTP DATUM)
                   (EQ 2 (LENGTH DATUM]
)
)

(DECLARE: EVAL@COMPILE

(PUTPROPS SELF MACRO ((A B)
```

```
                        A))

(PUTPROPS STRICTEOF MACRO ((CLASS)
                           (EQ CLASS (QUOTE EOF))))

(PUTPROPS TCONC.FRONT MACRO [LAMBDA (PTR)                    (* Removes and returns the first element of a TCONC list;
                                                                returns NIL if empty.)
                                  (PROG1 (CAAR PTR)          (* Return the first element.)
                                      (if (EQ (CAR PTR)
                                              (CDR PTR))
                                          then
```

(* * If there is only one element in the list, the pointer to the tail of the list must be set to NIL.
If the list is empty, this will be a no-op.)

```
                                                (RPLACA PTR NIL)
                                                (RPLACD PTR NIL)
                                          else
```

(* * Otherwise remove the first element of the list (for n elements in list, n>=1.))

```
                                            (RPLACA PTR (CDAR PTR))))])
)
```

(* * State machine generation. For an explanation of the parser technology, see "Theory & Construction of LR(k) Parsers" (Preliminary Version), Benjamin M. Brosgol, Center for Research in Computing Technology, Harvard University, March 1973)

```
(DEFINEQ
```

## (PG.LR0FSM

```
  [LAMBDA (G)                                                (* hts: " 3-Apr-86 16:08")
```

(* * Builds and returns the "LR(0)-FSM" of grammar G.)

```
    (BLOCK)
```

(* * First build the "[A]-FSM" for each production in the grammar, and save them in the list NTSL
(whose CAR will be the start state of the root production of the grammar))

```
    (LET ((NTSL (PG.AFSMS G)))
```

(* * Connect the "[A]-FSMs" together. This is the so-called dotted-linking and -unlinking stage.)

```
         (LET [(M (PG.CONNECT.AFSMS NTSL (fetch (GRAMMAR SymbolTable) of G]
```

(* * Using the Rabin-Scott algorithm, remove nondeterminism from the FSM.)

```
             (PG.RABIN.SCOTT M])
```

## (PG.AFSMS

```
  [LAMBDA (G)                                                (* hts: " 3-Apr-86 16:08")
```

(* * Constructs the "[A]-FSM" for each non terminal by linking the productions together as single strand transition nets.
Returns a list of their start states, with the start state of the root production as the first element of that list.)

(* * Details: There is a single final state, FINAL, that all the "[A]-FSMs" share.
The start states should be accumulated in order so that the "LR(0)-FSM" and machines built from it will be more comprehensible to humans -- hence the use of TCONC. The hasharray mapping NAME.TO.START.STATE makes it faster to find the start state of the "[A]-FSM" associated with a given nonterminal
(faster than looking on the TCONC list)%. Finally, the "[A]-FSMs" had better all be deterministic.)

```
    (BLOCK)
    (for P in (CONS (PG.ROOTPRODUCTION G)
                    (fetch PRODUCTIONS of G))
       bind START FINAL START.STATES NAME.TO.START.STATE first (SETQ FINAL (create STATE
                                                                                    NAME _ (QUOTE FINAL)))
                                                               (SETQ START.STATES (CONS))
                                                               (SETQ NAME.TO.START.STATE (HASHARRAY 33))
       do (SETQ START (GETHASH (fetch LEFTHAND of P)
                               NAME.TO.START.STATE))
          (if (NOT (type? STATE START))
              then (SETQ START (create STATE
                                       NAME _ (fetch LEFTHAND of P)))
                   (TCONC START.STATES START)
                   (PUTHASH (fetch LEFTHAND of P)
                            START NAME.TO.START.STATE))
          (for A in (fetch ALTERNATIVES of P) do (PG.AFSM.ADDPROD (create PRODUCTION
                                                                          LHS _ (fetch LEFTHAND of P)
                                                                          RHS _ (fetch RULE of A)
                                                                          AUGMENT _ (fetch (ALTERNATIVE AUGMENT)
                                                                                           of A))
                                                                  START FINAL))
       finally (SETQ START.STATES (CAR START.STATES))          (* Get rid of TCONC cell)
               (if (NOT (for S in START.STATES always (PG.DETERMINISTIC S)))
```

```
            then (SHOULDNT "Nondeterminism"))
        (if (NEQ (QUOTE START)
                 (fetch NAME of (CAR START.STATES)))
            then (SHOULDNT))
        (RETURN START.STATES])
```

## (PG.ROOTPRODUCTION

```
[LAMBDA (G)                                                    (* hts: "28-Feb-86 17:45")
```

(* * Generates and returns a production rule which can serve as the root production -- it worries with the problem of EOF as terminating the parse.)

```
  (create FPRODUCTION
          LEFTHAND _ (QUOTE START)
          ALTERNATIVES _ (LIST (create ALTERNATIVE
                                       RULE _ (LIST (fetch StartSymbol of G)
                                                    (QUOTE EOF))
                                       AUGMENT _ NIL])
```

## (PG.AFSM.ADDPROD

```
[LAMBDA (PROD S FIN)                                           (* hts: " 3-Apr-86 16:08")
  (if (NEQ (fetch NAME of S)
           (fetch LHS of PROD))
      then (\ILLEGAL.ARG S))
  (if (NOT (type? STATE FIN))
      then (\ILLEGAL.ARG FIN))
```

(* * Adds the given production to S, creating new states if necessary, finally linking it in to the FINAL state. Builds new states in such a way as to ensure determinism.)

```
  (BLOCK)
  (for TOKEN in (fetch RHS of PROD) bind S2 L S3 first (SETQ S2 S)
     do (SETQ L (for L2 in (fetch OUT of S2) thereis (EQ (fetch SYM of L2)
                                                         TOKEN)))
        (SETQ S2 (if (type? LINK L)
                     then (CAR (fetch ST of L))
                   else (SETQ S3 (create STATE))
                        (PG.CONNECT S2 S3 TOKEN)
                        S3))
     finally (PG.CONNECT S2 FIN PROD))
  S])
```

## (PG.CONNECT

```
[LAMBDA (FROM.STATE TO.STATE TRANSITION.SYMBOL)                (* hts: " 3-Apr-86 16:08")
```

(* * Connects FROM.STATE to TO.STATE with a transition labelled TRANSITION.SYMBOL)

```
  (BLOCK)
  (replace OUT of FROM.STATE with (PG.ADD.ARC (fetch OUT of FROM.STATE)
                                              TO.STATE TRANSITION.SYMBOL])
```

## (PG.ADD.ARC

```
[LAMBDA (LINKS TO.STATE TRANSITION.SYMBOL)                     (* hts: "28-Feb-86 14:48")
```

(* * Augments the set of transitions LINKS by adding a transition via TRANSITION.SYMBOL to TO.STATE %. Returns the augmented transition sets. Note: tries to keep them in order of acquisition)

```
  (LET [(TRANSITION (for LINK in LINKS thereis (EQ (fetch SYM of LINK)
                                                   TRANSITION.SYMBOL]
       (if (type? LINK TRANSITION)
           then (if (FMEMB TO.STATE (fetch ST of TRANSITION))
                    then (fetch ST of TRANSITION)
                  else (push (fetch ST of TRANSITION)
                             TO.STATE))
                LINKS
         else (CONS (create LINK
                            SYM _ TRANSITION.SYMBOL
                            ST _ (LIST TO.STATE))
                    LINKS])
```

## (PG.DETERMINISTIC

```
[LAMBDA (S)                                                    (* hts: " 3-Apr-86 16:08")
  (if (type? FSM S)
      then (SETQ S (fetch START of S)))
```

(* * Returns T if S is a deterministic FSM, NIL otherwise. For a state to have deterministic transitions to the next state,
(a) there may not be more than one arc out with the same transition symbol, and
(b) there may not be more than one state reachable by any transition symbol.
(* * Details: HASHARRAY glop is to ensure you don't get into infinite loops because of cycles in the FSM. Could have implemented this by calling STATES, but that would CONS a lot more and require essentially two traversals instead of one. Recursion shouldn't hurt because these FSMs are more bushy than they are deep.))

```
    (BLOCK)
    (PG.DETERMINISTIC1 S (HASHARRAY 33])
```

## (PG.DETERMINISTIC1

```
  [LAMBDA (S DONE)                                                    (* hts: " 3-Apr-86 16:08")
```

(* * Tells whether the submachine beginning at S is deterministic.
See comments in DETERMINISTIC.)

```
    (OR (GETHASH S DONE)
        (PROGN (PUTHASH S T DONE)
               (for ARC in (fetch OUT of S) bind (SYMBOLS _ NIL)
                  always (AND (PROG1 (NOT (FMEMB (fetch SYM of ARC)
                                                 SYMBOLS))
                                     (push SYMBOLS (fetch SYM of ARC)))
                              (PG.SINGLETON (fetch ST of ARC))
                              (PG.DETERMINISTIC1 (CAR (fetch ST of ARC))
                                     DONE])
```

## (PG.CONNECT.AFSMS

```
  [LAMBDA (NTSL SYMBOL.TABLE)                                         (* hts: " 3-Apr-86 17:46")
```

(* * Interconnects all the "[A]-FSMs" This is the dotted linking and unlinking phase.
NTSL is a list of start states of the various "[A]-FSMs" , with the overall start state first.
SYMBOL.TABLE maps symbols onto {TERMINAL, NONTERMINAL}.)

```
    (LET [(START (HASHARRAY (LENGTH NTSL)))
          (DONE (HASHARRAY (TIMES 5 (LENGTH NTSL]
```

(* * Build a hasharray which implements a fast mapping: name of the nonterminal to which an "[A]-FSM" reduces -> the start state of that "[A]-FSM")

```
         (for S in NTSL do (PUTHASH (fetch NAME of S)
                                    S START))
```

(* * Absorb to each state all the links of any machine which reduces to a nonterminal which the state is supposed to read)

```
         (for S in (PG.STATES NTSL) do (PG.CONNECT.AFSMS1 S (LIST S)
                                                SYMBOL.TABLE START DONE))
```

(* * Check to make sure everything got connected up properly.)

```
         (PG.CONNECTED NTSL SYMBOL.TABLE START))
```

(* * Start state better be first in the list)

```
    (if (NEQ (QUOTE START)
             (fetch NAME of (CAR NTSL)))
        then (SHOULDNT "Start state in wrong place"))
```

(* * Return the fully-connected state machine)

```
    (create FSM
            START _ (CAR NTSL)
            SymbolTable _ SYMBOL.TABLE])
```

## (PG.CONNECT.AFSMS1

```
  [LAMBDA (S PATH SYMBOL.TABLE START DONE)                            (* hts: " 4-Apr-86 16:29")
```

(* * For each transition out of S, if it is a transition on a nonterminal
(say A) and is not immediately reentrant (ie, the first state of the "[A]-FSM" for A -> Ax for some x), then S inherits all the out-links of the "[A]-FSM" (for A)%. The inheritance is done depth-first, since the "[A]-FSM" may have a first transition on a nonterminal, and so may need to inherit some transitions itself.
When a state has been completed, it is entered in the hashtable DONE;
if a state is already in this table, it can have links copied from it directly, and does not have its inheritance checked again.
Of course, if there are cyclically left-recursive rules (eg, A -> Bx, B -> Ay) then this depth first search could result in a cycle.
To avoid this, the routine remembers and checks its current search chain.
If a cycle is detected, it is returned and not dealt with. If the caller's state is not involved in the cycle, he deals with it (see CONNECT.CYCLES); else he returns it to his caller.)

```
    (if (NOT (GETHASH S DONE))
        then (for TRANSITION in (fetch OUT of S) bind SYMBOL AFSM CYCLE NONTERMINAL (COMPLETED _ T)
                do (SETQ SYMBOL (fetch SYM of TRANSITION))
                   (SETQ NONTERMINAL (PG.NONTERMINALP SYMBOL SYMBOL.TABLE))
                   (SETQ CYCLE (FMEMB SYMBOL PATH))
                   (SETQ AFSM (GETHASH SYMBOL START))
                   (if (AND NONTERMINAL (NOT CYCLE)
                            (NEQ S AFSM))
                       then (PG.CONNECT.AFSMS1 AFSM (CONS SYMBOL PATH)
                                  SYMBOL.TABLE START DONE)
                            (if (NOT (GETHASH AFSM DONE))
                                then (SETQ COMPLETED NIL))
                            [for TRANSITION2 in (fetch OUT of AFSM)
                               do (for S2 in (fetch ST of TRANSITION2) do (PG.CONNECT S S2 (fetch SYM of TRANSITION2
```

```
                                                                                ]
                    elseif (AND NONTERMINAL CYCLE (NEQ S AFSM))
                      then (SETQ COMPLETED NIL))
                finally (if COMPLETED
                            then (PUTHASH S T DONE])
```

## (PG.SINGLETON

```
[LAMBDA (LST)                                                   (* hts: "16-Feb-86 13:48")

        (* * Returns T if LST is a singleton list, NIL otherwise.)

  (AND (LISTP LST)
       (NOT (CDR LST])
```

## (PG.CONNECTED

```
[LAMBDA (NTSL SYMBOL.TABLE START)                               (* hts: " 3-Apr-86 16:09")

        (* * Checks to see if CONNECT.AFSMS has done its work correctly.
        calls SHOULDNT iff it hasnt.)

  (for S in (PG.STATES NTSL) do (for L in (fetch OUT of S)
                                   do (if (PG.NONTERMINALP (fetch SYM of L)
                                                           SYMBOL.TABLE)
                                          then (OR (PG.OWNS.LINKS S (GETHASH (fetch SYM of L)
                                                                             START))
                                                   (SHOULDNT "Ownership fault"])
```

## (PG.OWNS.LINKS

```
[LAMBDA (S1 S2)                                                 (* hts: "21-Feb-86 17:10")

        (* * Returns non-NIL iff S1s transitions are a superset of S2s.)

  (BLOCK)
  (for L2 in (fetch OUT of S2) always (for NEXT2 in (fetch ST of L2)
                                         always (for L1 in (fetch OUT of S1)
                                                   thereis (AND (EQ (fetch SYM of L2)
                                                                    (fetch SYM of L1))
                                                                (for NEXT1 in (fetch ST of L1)
                                                                   thereis (EQ NEXT1 NEXT2])
```

## (PG.RABIN.SCOTT

```
[LAMBDA (M)                                                     (* hts: " 3-Apr-86 16:08")

        (* * Rabin-Scott algorithm for making deterministic a FSM. Returns the determinized FSM.)

  (PG.RABIN.SCOTT1 (fetch START of M)
          (HASHARRAY 33)
          (HASHARRAY 33))
  (if (NOT (PG.DETERMINISTIC M))
      then (SHOULDNT "Nondeterminism"))
  M])
```

## (PG.RABIN.SCOTT1

```
[LAMBDA (S STATES DONE)                                         (* hts: " 3-Apr-86 16:08")

        (* * Does the actual work of the Rabin-Scott determinizing algorithm.
        Tail recursively enumerates the states of the machine (including replacement states)%.
        Recursion should be ok here because FSMs should be more bushy than deep.
        Basically, if S has transitions to two or more states S1, ..., Sn on the same symbol, replaces those transitions with a single
        transition to a union state (which has all the out transitions of S1 thru Sn)%.
        DONE hashtable has an entry for each state visited, so that you can detect cycles and not go catatonic over them.
        STATES hashtable maps replacement state names onto union states;
        this is used to avoid unnecessary duplication of union states)

  (LET ((S.NAME (fetch NAME of S)))
       (if (GETHASH S.NAME DONE)
           then S
        else (PUTHASH S.NAME T DONE)
             (for SYM in (PG.TRANSITION.SYMBOLS S) bind NEXT PG.REPLACEMENT
                do (SETQ NEXT (PG.NEXT.STATES S SYM))
                   (SETQ PG.REPLACEMENT (PG.REPLACEMENT NEXT STATES))
                   (if PG.REPLACEMENT
                       then (for N inside NEXT do (PG.DISCONNECT S N SYM))
                            (PG.CONNECT S PG.REPLACEMENT SYM)))
             (for L in (fetch OUT of S) do (OR (PG.SINGLETON (fetch ST of L))
                                               (SHOULDNT "Nondeterministic state"))
                                           (PG.RABIN.SCOTT1 (CAR (fetch ST of L))
                                                  STATES DONE))
             S])
```

## (PG.REPLACEMENT

```
  [LAMBDA (S STATES)                                                    (* hts: " 3-Apr-86 16:08")

          (* * Finds or generates the union state (if any) for the list of states S.)

    (if (AND (LISTP S)
             (CDR S))
        then (LET* ((NAME (PG.COMPOSITE.STATE.NAME S))
                    (PG.REPLACEMENT (GETHASH NAME STATES)))
                   (if (NOT (type? STATE PG.REPLACEMENT))
                       then (SETQ PG.REPLACEMENT (create STATE
                                                         NAME _ NAME))
                            (for SYM in (PG.TRANSITION.SYMBOLS S) do (for NEXT inside (PG.NEXT.STATES S SYM)
                                                                                      do (PG.CONNECT PG.REPLACEMENT NEXT SYM))
                                )
                            (PUTHASH NAME PG.REPLACEMENT STATES))
                   PG.REPLACEMENT)
      else NIL])
```

## (PG.TRANSITION.SYMBOLS

```
  [LAMBDA (STATESET)                                                    (* hts: "25-Feb-86 23:19")

          (* * Finds all the transition symbols that come from any of the states in STATESET)

    (LET ((SYMS (CONS)))
         [for S inside STATESET do (for L in (fetch OUT of S)
                                      do (if (NOT (FMEMB (fetch SYM of L)
                                                         (CAR SYMS)))
                                             then (TCONC SYMS (fetch SYM of L]
         (CAR SYMS])
```

## (PG.NEXT.STATES

```
  [LAMBDA (STATESET SYMBOL)                                             (* hts: " 3-Apr-86 16:08")
    (LET [(NEXT (for S inside STATESET bind TRANSITION
                   when [type? LINK (SETQ TRANSITION (for L in (fetch OUT of S)
                                                        thereis (EQ SYMBOL (fetch SYM of L]
                   join (COPY (fetch ST of TRANSITION]
         (if (PG.SINGLETON NEXT)
             then (CAR NEXT)
           else NEXT])
```

## (PG.COMPOSITE.STATE.NAME

```
  [LAMBDA (STATESET)                                                    (* hts: "25-Feb-86 23:15")

          (* * Gives a unique name to the set of states STATESET)

    (if (NLISTP STATESET)
        then (fetch NAME of STATESET)
      else (PACK (SORT (for S in STATESET collect (fetch NAME of S))
                       (FUNCTION ALPHORDER])
```

## (PG.DISCONNECT

```
  [LAMBDA (A B TRANSITION.SYMBOL)                                       (* hts: " 3-Apr-86 16:08")

          (* * Breaks the link between states A and B along TRANSITION.SYMBOL)

    (BLOCK)
    (LET [(TRANSITION (for LINK in (fetch OUT of A) thereis (EQ TRANSITION.SYMBOL (fetch SYM of LINK]
         [if (type? LINK TRANSITION)
             then (if (PG.SINGLETON (fetch ST of TRANSITION))
                      then

          (* this is the only transition that is keeping the link alive, so kill the link)

                           (replace OUT of A with (DREMOVE TRANSITION (fetch OUT of A)))
                  else                                                  (* chop B out of the link)
                       (replace ST of TRANSITION with (DREMOVE B (fetch ST of TRANSITION]
         (QUOTE DISCONNECTED])
```

## (PG.NONTERMINALP

```
  [LAMBDA (SYM SYMBOL.TABLE)                                            (* hts: " 1-Mar-86 17:06")

          (* * Returns T if SYM is a nonterminal symbol according to SYMBOL.TABLE , which was built from the current grammar)

    (EQ (GETHASH SYM SYMBOL.TABLE)
        (QUOTE NONTERMINAL])
```

## (PG.TERMINALP

```
  [LAMBDA (SYM SYMBOL.TABLE)                                            (* hts: "24-Feb-86 16:28")

          (* * Determines if SYM is a terminal for the current grammar. NIL is a terminal for all grammars.)
```

```
    (OR (EQ (GETHASH SYM SYMBOL.TABLE)
            (QUOTE TERMINAL))
        (NULL SYM])

)

(DEFINEQ
```

## (PG.LALRKFSM

```
  [LAMBDA (FSM)                                                    (* hts: " 3-Apr-86 16:09")

          (* Transforms an "LR(0)-FSM" into a "LALR(K)-FSM" by adding lookahead states to resolve inadequate states of the
          "LR(0)-FSM" %. Also builds inverse transition links; the backlinks, in addition to helping determine the lookahead sets, help
          in building the actual parser (because you have to backtrack states on reduction))

    (LET ((PG.STATES (PG.STATES FSM))
          (SYMBOL.TABLE (fetch (FSM SymbolTable) of FSM)))

          (* * Build inverse transition links.)

          (PG.BUILD.BackLinks PG.STATES)

          (* * Resolve all inadequate states by adding lookahead states.)

          (for S in PG.STATES do (SELECTQ (PG.STATETYPE S SYMBOL.TABLE)
                                          (INADEQUATE (PG.PRINT.INADEQUATE S SYMBOL.TABLE)
                                                      (PG.RESOLVE S SYMBOL.TABLE))
                                          ((READ REDUCE LOOKAHEAD)
                                              NIL)
                                          (NIL (if (NEQ (fetch NAME of S)
                                                        (QUOTE FINAL))
                                                   then (SHOULDNT "Null state type")))
                                          (SHOULDNT "Queer state type")))

          (* * Make sure there are no remaining inadequate states.)

          (if (for S in (PG.STATES FSM) thereis (EQ (PG.STATETYPE S SYMBOL.TABLE)
                                                    (QUOTE INADEQUATE)))
              then (SHOULDNT "Inadequate states remaining")))
    FSM])
```

## (PG.BUILD.BackLinks

```
  [LAMBDA (STATES)                                                 (* hts: " 3-Apr-86 16:08")

          (* * Puts backward links on each of the state S in FSM, showing which from which states you could have arrived at S and by
          what token transition.)

    [for S1 in STATES do (for L in (fetch OUT of S1) do (for S2 in (fetch ST of L)
                                                               do (PG.CONNECT.BACK S2 S1 (fetch SYM of L]
    (QUOTE BACKLINKED])
```

## (PG.CONNECT.BACK

```
  [LAMBDA (FROM.STATE TO.STATE TRANSITION.SYMBOL)                  (* hts: " 3-Apr-86 16:08")

          (* * Connects FROM.STATE to TO.STATE with a backwards transition labelled TRANSITION.SYMBOL)

    (BLOCK)
    (replace BackLinks of FROM.STATE with (PG.ADD.ARC (fetch BackLinks of FROM.STATE)
                                                      TO.STATE TRANSITION.SYMBOL))
    (QUOTE CONNECTED-BACKWARDS])
```

## (PG.STATETYPE

```
  [LAMBDA (S SYMBOL.TABLE)                                         (* hts: " 3-Apr-86 16:09" posted: "20-MAY-77 23:03")

          (* * Examines a state and determines its type for the parser.)

    (for L in (fetch OUT of S) bind (TYPE _ NIL) do [COND
                                                    ((PG.LOOKAHEADP L)
                                                     (SELECTQ TYPE
                                                          ((NIL LOOKAHEAD)
                                                               (SETQQ TYPE LOOKAHEAD))
                                                          (SHOULDNT)))
                                                    ((type? PRODUCTION (fetch SYM of L))
                                                     (SELECTQ TYPE
                                                          ((READ REDUCE INADEQUATE)
                                                               (SETQQ TYPE INADEQUATE))
                                                          (NIL (SETQQ TYPE REDUCE))
                                                          (SHOULDNT)))
                                                    ((PG.TERMINALP (fetch SYM of L)
                                                                   SYMBOL.TABLE)
                                                     (SELECTQ TYPE
                                                          ((READ NIL)
                                                               (SETQQ TYPE READ))
                                                          ((REDUCE INADEQUATE)
```

```
                                                    (SETQQ TYPE INADEQUATE))
                                                (SHOULDNT]
        finally (RETURN TYPE])
```

## (PG.RESOLVE

```
[LAMBDA (S SYMBOL.TABLE)                                    (* hts: "16-Apr-86 15:26")
```

(* * Attempts to resolve inadequacy in a state. Note that if the language is not "LALR(k)" for any k, this routine will go into an infinite loop; but it will announce its progress to the user so he can
(a) stop it if it looks runaway, and (b) know what the conflict is.)

```
    (LET ((LOOKAHEAD (PG.BUILD.LOOKAHEAD.SETS S SYMBOL.TABLE)))
         (for LOOK in LOOKAHEAD bind EXTRA.STATE LINK SYMBOL DEST LLA
            do (SETQ LINK (CAR LOOK))
               (SETQ SYMBOL (fetch SYM of LINK))
               (SETQ DEST (CAR (fetch ST of LINK)))
               (SETQ LLA (CDR LOOK))
               (SETQ EXTRA.STATE (create STATE))
               (PG.CONNECT EXTRA.STATE DEST SYMBOL)
               (for SYMS in LLA do (PG.CONNECT S EXTRA.STATE SYMS))
               (replace BackLinks of EXTRA.STATE with S)
               (PG.DISCONNECT S DEST SYMBOL)))
    (QUOTE RESOLVED])
```

## (PG.PRINT.INADEQUATE

```
[LAMBDA (S SYMBOL.TABLE)                                    (* hts: " 3-Apr-86 16:08")
```

(* * Prints out information about the inadequate state about to be resolved.
Tells what transitions are possible from this state, thus showing the conflict.
Should the parser generator enter an infinite loop trying to resolve this state
(ie, if the language is not "LALR(k)" for any k), this will help the user find the rules in his grammar which are responsible for the problem.)

```
    (PRINTOUT NIL "Adding lookahead to resolve conflict in state " (fetch NAME of S)
           ":" T)
    (for L in (fetch OUT of S) do (LET ((SYM (fetch SYM of L)))
                                       (if (PG.TERMINALP SYM SYMBOL.TABLE)
                                           then (PRINTOUT NIL "   Read:   " SYM T)
                                         elseif (type? PRODUCTION SYM)
                                           then (PRINTOUT NIL "   Reduce: " (fetch LHS of SYM)
                                                        " -> ")
                                                (for THING in (fetch RHS of SYM) do (PRINTOUT NIL THING " "))
                                                (PRINTOUT NIL T])
```

## (PG.BUILD.LOOKAHEAD.SETS

```
[LAMBDA (S SYMBOL.TABLE)                                    (* hts: "16-Apr-86 15:26")
```

(* * Attempts to resolve inadequacy in a state. Note that if the language is not "LALR(k)" for any k, this routine will go into an infinite loop; but it will announce its progress to the user so he can
(a) stop it if it looks runaway, and (b) know what the conflict is. If possible, returns a list of things of the form
(link %. set of lookahead strings) such that each link from S is given a set of lookahead strings disjoint from any belonging to another of S's links.)

```
    (for K from 1 bind LLAS do
```

(* * Tell user how deep you're going, so he can detect runaway
(in case the grammar he gave is more complex than he expected))

```
                                (PRINTOUT NIL "   " K "-level lookahead from state " (fetch NAME of S)
                                       T)
                                [SETQ LLAS (for L in (fetch OUT of S)
                                              when (NOT (PG.NONTERMINALP (fetch SYM of L)
                                                                         SYMBOL.TABLE))
                                              collect (CONS L (PG.LLA K (LIST S)
                                                                      L SYMBOL.TABLE NIL]
                                (if (PG.DISJOINT LLAS)
                                    then (RETURN LLAS])
```

## (PG.DISJOINT

```
[LAMBDA (LLASET)                                            (* hts: " 1-Mar-86 21:59")
```

(* * Tells whether the lookahead sets in LLASET are all disjoint.
Note that entries on LLASET are of the form (link . set-of-lookahead-chars))

```
    (for L on LLASET bind L1 never (SETQ L1 (CAR L))
                                   (for L2 in (CDR L) thereis (INTERSECTION (CDR L1)
                                                                            (CDR L2])
```

## (PG.LLA

```
[LAMBDA (K PATH LINK SYMBOL.TABLE ALREADY)                  (* hts: " 3-Apr-86 16:09")
```

(* * Finds the level-K set of lookahead symbols generated by looking along the state path PATH
(which is backwards)%. This local lookahead set is used to build the next level of lookahead states to resolve an inadequate

state. ALREADY is a set of states already visited, and it is intended to prevent infinite recursion because of cycles.)

```
(COND
   ((LEQ K 0)

      (* * The lookahead set for lookahead strings of length 0 is the set containing the empty string.
      This stops the recursion.)

       (LIST NIL))
      ((LISTP (fetch SYM of LINK))
       (PG.LLA.LOOKAHEAD K PATH LINK SYMBOL.TABLE ALREADY))
      ((PG.TERMINALP (fetch SYM of LINK)
            SYMBOL.TABLE)
       (PG.LLA.READ K PATH LINK SYMBOL.TABLE ALREADY))
      ((type? PRODUCTION (fetch SYM of LINK))
       (PG.LLA.REDUCE K PATH LINK SYMBOL.TABLE ALREADY))
      (T (\ILLEGAL.ARG LINK])
```

## (PG.LLA.LOOKAHEAD

```
[LAMBDA (K PATH LINK SYMBOL.TABLE ALREADY)                    (* hts: " 3-Apr-86 16:08")

      (* * Lookahead transition. Skip over it.)

   (LET* [(NEXT.STATE (CAR (fetch ST of LINK)))
          (NEXT.LOOKAHEAD (for L in (fetch OUT of NEXT.STATE) when (NOT (PG.NONTERMINALP (fetch SYM of L)
                                                                                SYMBOL.TABLE))
                                  join (PG.LLA K PATH L SYMBOL.TABLE ALREADY]
          (INTERSECTION NEXT.LOOKAHEAD NEXT.LOOKAHEAD])
```

## (PG.LLA.READ

```
[LAMBDA (K PATH LINK SYMBOL.TABLE ALREADY)                    (* hts: " 3-Apr-86 16:08")

      (* * Read transition. Current lookahead symbol, obviously, is the transition symbol of the current link.
      Recurse to find deeper lookahead. (Except if current symbol is EOF, in which case all deeper lookahead must be EOF also,
      by definition of LLA.))

   (if (EQ (QUOTE EOF)
           (fetch SYM of LINK))
       then (LIST (to K collect (QUOTE EOF)))
     else (LET* [(NEXT.STATE (CAR (fetch ST of LINK)))
                 (NEXT.LOOKAHEAD (for L in (fetch OUT of NEXT.STATE)
                                    when (NOT (PG.NONTERMINALP (fetch SYM of L)
                                                  SYMBOL.TABLE))
                                    join (for LOOK in (PG.LLA (SUB1 K)
                                                              (CONS NEXT.STATE PATH)
                                                              L SYMBOL.TABLE ALREADY)
                                            collect (CONS (fetch SYM of LINK)
                                                          (COPY LOOK]
          (INTERSECTION NEXT.LOOKAHEAD NEXT.LOOKAHEAD])
```

## (PG.LLA.REDUCE

```
[LAMBDA (K PATH LINK SYMBOL.TABLE ALREADY)                    (* hts: " 3-Apr-86 16:09")

      (* * Reduce transition: Back up along symbols of right-hand side of reduction rule %.
      Then follow the symbol of the left hand side to get to another state.
      Now we've essentially performed the reduction and got back in the FSM to where we would have been if we had just read
      the nonterminal symbol (if such were possible)%. Lookahead symbol search can proceed
      (recursively) from there.)

   (LET* [(PRODUCTION (fetch SYM of LINK))
          (LHS (fetch LHS of PRODUCTION))
          (RHS (fetch RHS of PRODUCTION))
          (PG.BACKUP (PG.BACKUP PATH (REVERSE RHS)))
          (SHORTENED.PATH (FNTH PATH (PLUS 2 (LENGTH RHS]
         (LET [(LOOK (for S in PG.BACKUP
                        join (LET ((TARGET (PG.REDUCE.TARGET S LHS)))
                                  (if (FMEMB TARGET ALREADY)
                                      then NIL
                                    else (for L in (fetch OUT of TARGET)
                                                when (NOT (PG.NONTERMINALP (fetch SYM of L)
                                                                  SYMBOL.TABLE))
                                                join (PG.LLA K (CONS TARGET (CONS S SHORTENED.PATH))
                                                             L SYMBOL.TABLE (CONS TARGET ALREADY]
              (INTERSECTION LOOK LOOK])
```

## (PG.BACKUP

```
[LAMBDA (STATE.PATH SYMBOL.PATH)                              (* hts: "17-Apr-86 16:25" posted: "23-JUN-77 13:45")

      (* * Designates the set of states along PATH from which you could have reached
      (CAR PATH) by reading the symbols in SYMBOL.PATH. STATE.PATH is a
      (perhaps incomplete) path of states along which to back up; (CAR STATE.PATH) is the state from which to start backing.
      SYMBOL.PATH is the list of symbols along which to back up (empty in the case of rules like A -> e)%.)
```

```
    (SETQ STATE.PATH (for S in STATE.PATH collect (PG.LOOKAHEAD.SOURCE S)))

          (* * Back up as far as you are fully constrained by the symbol path provided.)

    (while (AND SYMBOL.PATH (CDR STATE.PATH))
       do (if [NOT (for LNK in (fetch BackLinks of (CAR STATE.PATH))
                      thereis (AND (EQ (fetch SYM of LNK)
                                       (CAR SYMBOL.PATH))
                                   (FMEMB (CADR STATE.PATH)
                                          (fetch ST of LNK]
              then (SHOULDNT))
           (SETQ STATE.PATH (CDR STATE.PATH))
           (SETQ SYMBOL.PATH (CDR SYMBOL.PATH)))

          (* * Back up the rest of the way (if any)%. This may produce some fanning out.)

    (bind (STATE.SET _ (LIST (CAR STATE.PATH))) while SYMBOL.PATH
       do (for S in STATE.SET bind (STATES _ NIL)
             do (for PREV in [fetch ST of (for L in (fetch BackLinks of S) thereis (EQ (fetch SYM of L)
                                                                                       (CAR SYMBOL.PATH]
                   do (if (NOT (FMEMB PREV STATES))
                          then (SETQ STATES (NCONC1 STATES PREV)))
                   finally (SETQ STATE.SET STATES)))
           (SETQ SYMBOL.PATH (CDR SYMBOL.PATH))
       finally (RETURN STATE.SET])
```

## (PG.LOOKAHEAD.SOURCE

```
  [LAMBDA (S)                                                          (* hts: "20-Feb-86 11:53")

          (* * If S was generated to do lookahead, you really want to start backign up from the state that S came from.
          States generated to do lookahead always have their source (the state from which they do lookahead) in their BackLinks
          field; other states have lists of states in their BackLinks field.)

    (if (type? STATE (fetch BackLinks of S))
        then (fetch BackLinks of S)
      else S])
```

## (PG.REDUCE.TARGET

```
  [LAMBDA (STATE NONTERMINAL)                                          (* hts: "24-Feb-86 18:59")

          (* * Returns the state connected by a transition from STATE along the nonterminal symbol NONTERMINAL)

    (CAR (fetch ST of (for LNK in (fetch OUT of STATE) thereis (EQ NONTERMINAL (fetch SYM of LNK])
```

## (PG.LOOKAHEADP

```
  [LAMBDA (L)                                                          (* hts: "24-Feb-86 15:53")

          (* * Tells whether the given link is a lookahead link. Ordinary backlinks fields contain lists of links;
          states generated by lookahead have just a state as their backlink: the state from which the lookahead was generated.)

    (type? STATE (fetch BackLinks of (CAR (fetch ST of L])

)

(DEFINEQ

## (PG.OLD.LLA

  [LAMBDA (K PATH LINK FIRST.LOOKAHEAD.SYMBOLS SYMBOL.TABLE ALREADY)
                                                                       (* hts: " 3-Apr-86 16:09")
    (if (NOT (AND (FIXP K)
                  (GREATERP K 0)))
        then (\ILLEGAL.ARG K))
    (if (NOT (OR (type? PRODUCTION (fetch SYM of LINK))
                 (PG.TERMINALP (fetch SYM of LINK)
                        SYMBOL.TABLE)))
        then (\ILLEGAL.ARG LINK))
    (if [NOT (AND (OR (NULL FIRST.LOOKAHEAD.SYMBOLS)
                      (LISTP FIRST.LOOKAHEAD.SYMBOLS))
                  (EQ (LENGTH FIRST.LOOKAHEAD.SYMBOLS)
                      (SUB1 K]
        then (\ILLEGAL.ARG FIRST.LOOKAHEAD.SYMBOLS))          (* hts: "21-Feb-86 13:41")

          (* * Finds the level-K set of lookahead symbols generated by looking along the state path PATH
          (which is backwards), given that you've already found the lookahead symbols FIRST.LOOKAHEAD.SYMBOLS
          (K-1th, ..., 1st.)%. This local lookahead set is used to build the next level of lookahead states to resolve an inadequate state.
          ALREADY is a set of states already visited, and it is intended to prevent infinite recursion because of cycles.)

    (LET [(PG.OLD.LLA
           (if (NOT (type? PRODUCTION (fetch SYM of LINK)))
               then

          (* * Read transition: the lookahead is just what you're about to read.)

                    (if (EQ K 1)
```

```
                    then

        (* * Terminal case: just return a list containing the symbol in the current transition)

                        (LIST (fetch SYM of LINK))
                else

        (* * Got to recurse to find the desired level of lookahead -- constrained by the lookahead symbols already found,
        FIRST.LOOKAHEAD.SYMBOLS.)

                    (if (EQ (fetch SYM of LINK)
                            (CAR FIRST.LOOKAHEAD.SYMBOLS))
                        then (LET [(NEXT.STATE (CAR (fetch ST of LINK]
                                  (for NEXT.LINK in (fetch OUT of NEXT.STATE)
                                     when (NOT (PG.NONTERMINALP (fetch SYM of NEXT.LINK)
                                                                SYMBOL.TABLE))
                                     join (PG.OLD.LLA (SUB1 K)
                                                      (CONS NEXT.STATE PATH)
                                                      NEXT.LINK
                                                      (CDR FIRST.LOOKAHEAD.SYMBOLS)
                                                      SYMBOL.TABLE ALREADY)))
                             else NIL))
            else

        (* * Reduce transition: Back up along symbols of right-hand side of reduction rule
        (constrained insofar as possible by the state path PATH)%. Then follow the symbol of the left hand side to get to another
        state. Now we've essentially performed the reduction and got back in the FSM to where we would have been if we had just
        read the nonterminal symbol (if such were possible)%. Lookahead symbol search can proceed
        (recursively) from there.)

                (LET* [(PRODUCTION (fetch SYM of LINK))
                       (LHS (fetch LHS of PRODUCTION))
                       (RHS (fetch RHS of PRODUCTION))
                       (PG.BACKUP (PG.BACKUP PATH (REVERSE RHS)))
                       (SHORTENED.PATH (FNTH PATH (PLUS 2 (LENGTH RHS]
                      (for S in PG.BACKUP
                         join (LET* ((TARGET (PG.REDUCE.TARGET S LHS)))
                                    (if (FMEMB TARGET ALREADY)
                                        then NIL
                                      else (for L in (fetch OUT of TARGET)
                                              when (NOT (PG.NONTERMINALP (fetch SYM of L)
                                                                         SYMBOL.TABLE))
                                              join (PG.OLD.LLA K (CONS TARGET (CONS S SHORTENED.PATH))
                                                               L FIRST.LOOKAHEAD.SYMBOLS SYMBOL.TABLE (CONS TARGET
                                                                                                            ALREADY]
             (INTERSECTION PG.OLD.LLA PG.OLD.LLA])

)

(DECLARE: EVAL@COMPILE DONTCOPY

(DECLARE: EVAL@COMPILE

(DATATYPE PRODUCTION (LHS                                   (* an atom, the name of the nonterminal)
                          RHS                               (* a list of atoms, the right hand side of this rule)
                          AUGMENT                           (* name of a semantic action to be performed on reducing to this
                          rule)
                          ))

(DATATYPE FSM (START                                        (* Start state of the state machine)
                    SymbolTable                             (* a hashtable mapping symbols onto {terminal, nonterminal})
                    ))

(DATATYPE STATE (NAME                                       (* name of this state; an atom)
                     OUT                                    (* transitions from this state; a list of LINKs)
                     BackLinks                              (* how you could have got to this state)
                     )
       NAME _ (GENSYM))

(DATATYPE LINK (SYM

        (* thing to read or reduction rule for this transition. Either an ATOM or a PRODUCTION)

                    ST                                      (* list of states to which this transition brings you.
                                                            One element if deterministic)
                    ))
)

(/DECLAREDATATYPE (QUOTE PRODUCTION)
       (QUOTE (POINTER POINTER POINTER))
       ;; ---field descriptor list elided by lister---
       (QUOTE 6))

(/DECLAREDATATYPE (QUOTE FSM)
       (QUOTE (POINTER POINTER))
       ;; ---field descriptor list elided by lister---
```

```
       (QUOTE 4))

(/DECLAREDATATYPE (QUOTE STATE)
       (QUOTE (POINTER POINTER POINTER))
       ;; ---field descriptor list elided by lister---
       (QUOTE 6))

(/DECLAREDATATYPE (QUOTE LINK)
       (QUOTE (POINTER POINTER))
       ;; ---field descriptor list elided by lister---
       (QUOTE 4))
)

(/DECLAREDATATYPE (QUOTE PRODUCTION)
       (QUOTE (POINTER POINTER POINTER))
       ;; ---field descriptor list elided by lister---
       (QUOTE 6))

(/DECLAREDATATYPE (QUOTE FSM)
       (QUOTE (POINTER POINTER))
       ;; ---field descriptor list elided by lister---
       (QUOTE 4))

(/DECLAREDATATYPE (QUOTE STATE)
       (QUOTE (POINTER POINTER POINTER))
       ;; ---field descriptor list elided by lister---
       (QUOTE 6))

(/DECLAREDATATYPE (QUOTE LINK)
       (QUOTE (POINTER POINTER))
       ;; ---field descriptor list elided by lister---
       (QUOTE 4))

(DEFINEQ

(PG.SMASH.FSM
  [LAMBDA (M)                                          (* hts: " 3-Apr-86 16:09")

          (* * Kills circularity in the finite state machine M so that the silly refcount garbage collector can collect it.)

    (for S in (PG.STATES M) do (replace BackLinks of S with NIL))
    M])

)

(DEFINEQ

(PG.PPL
  [LAMBDA (L)                                          (* hts: " 3-Apr-86 16:09")

          (* * Prints where the link L leads and via what.)

    (printout NIL .TAB0 10 "(")
    (if (type? PRODUCTION (fetch SYM of L))
        then (PG.PPP (fetch SYM of L))
      else (printout NIL (fetch SYM of L)))
    (printout NIL ";")
    (for I in (fetch ST of L) do (printout NIL , (fetch NAME of I)))
    (printout NIL ")" T)
    L])

(PG.PPM
  [LAMBDA (M)                                          (* hts: " 3-Apr-86 16:09")

          (* * Prints out a representation of the finite state machine M)

    (for S in (SORT (PG.STATES M)
                    (FUNCTION PG.STATE.ORDER))
       do (PG.PPS S))
    (TERPRI)
    M])

(PG.PPP
  [LAMBDA (P)                                          (* hts: "25-Feb-86 00:14")

          (* * Prints out a representation of a production rule)

    (printout NIL (fetch LHS of P)
```

```
          " ->")
    (for TOKEN in (fetch RHS of P) do (PRINTOUT NIL " " TOKEN))
    P])
```

## (PG.PPS

```
  [LAMBDA (S)                                                    (* hts: " 3-Apr-86 16:09")

          (* * Prints crucial info about state S: transitions from it, whether it is a lookahead state, etc.)

    (printout NIL (fetch NAME of S))
    (if (for L in (fetch OUT of S) thereis (PG.LOOKAHEADP L))
        then (printout NIL .TAB 10 "Lookahead" T)
      else (printout NIL T))
    (if (fetch OUT of S)
        then (printout NIL "  Out:  ")
             (for L in (fetch OUT of S) do (PG.PPL L)))
    S])

)

(DEFINEQ
```

## (PG.STATES

```
  [LAMBDA (S)                                                    (* hts: " 3-Apr-86 18:06")
```

(* * Forms a list of the states of an FSM. Uses a hashtable (DONE) to determine whether a state has been visited. If not, it gets appended to the list of states. Recursion shouldn't hurt because these FSMs are more bushy than they are deep.)

```
    (BLOCK)
    (LET ((DONE (HASHARRAY 33))
          (STATES (CONS)))
         (COND
            ((type? FSM S)
             (PG.STATES1 (fetch START of S)
                    STATES DONE))
            ((LISTP S)
             (for S1 in S join (PG.STATES1 S1 STATES DONE)))
            (T (PG.STATES1 S STATES DONE)))
         (CAR STATES])
```

## (PG.STATES1

```
  [LAMBDA (S STATES DONE)                                        (* hts: "24-Feb-86 22:33")
```

(* * Collects substates of state S. See comments in STATES.)

```
    [if (NOT (GETHASH S DONE))
        then (TCONC STATES S)
             (PUTHASH S T DONE)
             (for ARC in (fetch OUT of S) do (for S2 in (fetch ST of ARC) do (PG.STATES1 S2 STATES DONE]
    NIL])
```

## (PG.STATE.ORDER

```
  [LAMBDA (S1 S2)                                                (* hts: "26-Feb-86 14:30")
```

(* * Returns T if S1's name should come before S2's; NIL otherwise)

```
    (LET ((S1-NAME (fetch NAME of S1))
          (S2-NAME (fetch NAME of S2)))
         (COND
            ((EQ S1-NAME (QUOTE START))
             T)
            ((EQ S2-NAME (QUOTE START))
             NIL)
            ((EQ S1-NAME (QUOTE FINAL))
             NIL)
            ((EQ S2-NAME (QUOTE FINAL))
             T)
            (T (ALPHORDER S1-NAME S2-NAME])

)
```

(* * Lisp code generation)

```
(DEFINEQ
```

## (PG.CODE.PARSER

```
  [LAMBDA (M PARSERSPEC SORTED?)                 (* hts: "16-Apr-86 15:31" posted: "20-MAY-77 23:01")
```

(* * Outputs a LISP program for the parser. PARSERSPEC is a parser specification, and M is the "LALR(k)-FSM" generated from that specification. Parser program structurally is just a great big PROG, with labels for each of the states in M and GOs to accomplish transitions from state to state. There are two stacks: USERSTACK contains the developing parse tree, and CONTROLSTACK keeps a record of the states that have been seen so far and so makes it possible to back up to the right place after reducing.)

```
    (LIST (QUOTE LAMBDA)
          (LIST (QUOTE EXPECTED)
                (QUOTE STATE))
          (LIST (QUOTE *)
                (QUOTE *)
                (QUOTE Parser)
                (fetch PARSERNAME of PARSERSPEC))
          (CONS (QUOTE PROG)
                (CONS (LIST (LIST (QUOTE CONTROLSTACK)
                                  (LIST (fetch STACKINITFN of PARSERSPEC)))
                            (LIST (QUOTE USERSTACK)
                                  (LIST (fetch STACKINITFN of PARSERSPEC)))
                            [LIST (QUOTE LOOKAHEAD)
                                  (LIST (fetch LAQUEUEINITFN of PARSERSPEC)
                                        (LIST (QUOTE CAR)
                                              (QUOTE STATE]
                            (LIST (QUOTE TEMP.LOOKAHEAD)
                                  (LIST (fetch QUEUEINITFN of PARSERSPEC)))
                            (QUOTE TOKEN)
                            (QUOTE CLASS)
                            (QUOTE LHS)
                            (QUOTE RHS))
                      (CONS (LIST (fetch PUSHFN of PARSERSPEC)
                                  (QUOTE CONTROLSTACK)
                                  (KWOTE (QUOTE START)))
                            (CONS (LIST (QUOTE GO)
                                        (QUOTE START))
                                  (PG.CODE.STATES M PARSERSPEC SORTED?])
```

## (PG.CODE.STATES

```
  [LAMBDA (M SPEC SORTED?)                                  (* hts: " 3-Apr-86 16:09")
```

(* * Generates code for each of the states in the stack-configuration-recognizing machine M.
Sorts the states by name if SORTED? is non-NIL. (This makes the code significantly easier for humans to read.))

```
    (for STATE in (if SORTED?
                      then (SORT (PG.STATES M)
                                 (FUNCTION PG.STATE.ORDER))
                    else (PG.STATES M))
       unless (EQ (fetch NAME of STATE)
                  (QUOTE FINAL))
       join (LET ((SYMBOL.TABLE (fetch (FSM SymbolTable) of M)))
                 (CONS (fetch NAME of STATE)
                       (SELECTQ (PG.STATETYPE STATE SYMBOL.TABLE)
                            (READ (PG.CODE.READ STATE SYMBOL.TABLE SPEC))
                            (REDUCE (PG.CODE.REDUCE STATE SPEC))
                            (LOOKAHEAD (PG.CODE.LOOKAHEAD STATE SYMBOL.TABLE SPEC))
                            (SHOULDNT])

)

(DEFINEQ
```

## (PG.CODE.LOOKAHEAD

```
  [LAMBDA (STATE SYMBOL.TABLE PARSERSPEC)                   (* hts: " 3-Apr-86 16:09")
```

(* * Generates the code to peek a token from the input stream and act on it appropriately.)

```
    (LIST (LIST (QUOTE *)
                (QUOTE Lookahead))
          (PG.CODE.LOOKAHEAD.ALL.TOKENS STATE SYMBOL.TABLE PARSERSPEC)
          (PG.CODE.LOOKAHEAD.SWITCH STATE SYMBOL.TABLE PARSERSPEC])
```

## (PG.CODE.LOOKAHEAD.ALL.TOKENS

```
  [LAMBDA (STATE SYMBOL.TABLE PARSERSPEC)                   (* hts: " 3-Apr-86 16:09")
```

(* * Generates the code to peek all the tokens necessary to do lookahead discrimination.
For 1-symbol lookahead, CLASS gets bound to the symbol; for n-symbol lookahead, n>1, CLASS gets bound to a list of symbols)

```
    (LET* [(LOOKAHEAD (for L in (fetch OUT of STATE) when (LISTP (fetch SYM of L))
                         collect (fetch SYM of L)))
           (NSYMS (LENGTH (CAR LOOKAHEAD]
          (if (EQ NSYMS 1)
              then
```

(* * CLASS is just the class of the token read; stick the token on the regular lookahead queue.)

```
                    (LIST (QUOTE SETQ)
                          (QUOTE CLASS)
                          (PG.CODE.LOOKAHEAD.TOKEN PARSERSPEC (for L in LOOKAHEAD collect (CAR L))
                                 (QUOTE LOOKAHEAD)))
            else
```

(* * If peeking multiple tokens, must first stick them on a special queue and then shift them onto the normal lookahead queue. This makes it possible for lookaheads to look at results of previous lookaheads without themselves looping on the same symbol.)

```
      (LIST (QUOTE PROG1)
            [LIST (QUOTE SETQ)
                  (QUOTE CLASS)
                  (CONS (QUOTE LIST)
                        (for N from 1 to NSYMS collect (PG.CODE.LOOKAHEAD.TOKEN
                                                         PARSERSPEC
                                                         (for L in LOOKAHEAD
                                                            collect (CAR (FNTH L N)))
                                                         (QUOTE TEMP.LOOKAHEAD]
            (LIST (QUOTE for)
                  (QUOTE TOKEN)
                  (QUOTE in)
                  (QUOTE CLASS)
                  (QUOTE do)
                  (LIST (fetch ENQUEUEFN of PARSERSPEC)
                        (QUOTE LOOKAHEAD)
                        (QUOTE TOKEN])
```

## (PG.CODE.LOOKAHEAD.TOKEN

```
[LAMBDA (PARSERSPEC EXPECTED QUEUE)                          (* hts: "28-Feb-86 21:59")
```

(* * Generates the code for peeking a token from the input stream.
Stores the thing on the lookahead buffer , determines its class and returns its class.)

```
   (LIST (QUOTE LET*)
         [LIST (LIST (QUOTE EXPECTED.OWN)
                     (KWOTE EXPECTED))
               (LIST (QUOTE TOKEN)
                     (LIST (QUOTE if)
                           (LIST (fetch QUEUENOTEMPTYFN of PARSERSPEC)
                                 (QUOTE LOOKAHEAD))
                           (QUOTE then)
                           (LIST (fetch DEQUEUEFN of PARSERSPEC)
                                 (QUOTE LOOKAHEAD))
                           (QUOTE else)
                           (LIST (fetch READFN of PARSERSPEC)
                                 (QUOTE EXPECTED.OWN)
                                 (LIST (QUOTE CDR)
                                       (QUOTE STATE]
         (LIST (fetch ENQUEUEFN of PARSERSPEC)
               QUEUE
               (QUOTE TOKEN))
         (LIST (fetch CLASSFN of PARSERSPEC)
               (QUOTE TOKEN)
               (QUOTE EXPECTED.OWN])
```

## (PG.CODE.LOOKAHEAD.SWITCH

```
[LAMBDA (STATE SYMBOL.TABLE PARSERSPEC)                      (* hts: " 3-Apr-86 16:09")
```

(* * Generates the code to make appropriate transitions from lookahead states.
Gotta watch out: EOF transitions must be checked last, in case of overly permissive EOF-acceptor fns.)

```
   (CONS (QUOTE COND)
         (NCONC1 [for L in [SORT (COPY (fetch OUT of STATE))
                                (FUNCTION (LAMBDA (L1 L2)
                                                  (GREATERP (for TOKEN in (fetch SYM of L1)
                                                               when (NEQ TOKEN (QUOTE EOF)) sum 1)
                                                            (for TOKEN in (fetch SYM of L2)
                                                               when (NEQ TOKEN (QUOTE EOF)) sum 1]
                           when (LISTP (fetch SYM of L)) collect (LIST (PG.CODE.LOOKAHEAD.MATCH (fetch SYM of L)
                                                                        PARSERSPEC)
                                                                  (LIST (QUOTE GO)
                                                                        (fetch NAME
                                                                           of (CAR (fetch ST of L]
                 (LIST T (LIST (QUOTE PARSE.ERROR])
```

## (PG.CODE.LOOKAHEAD.MATCH

```
[LAMBDA (LOOKAHEAD.PATH PARSERSPEC)                          (* hts: "28-Feb-86 17:46")
```

(* * Generates code which will be true if the tokens read match those in LOOKAHEAD.PATH, and false otherwise.
Matching is a little tricky, for two reasons: (a) if lookahead is only 1 token deep, it will be stored in CLASS bare
(ie, not wrapped in a list); and (b) you don't want to read eof more than once, and you want to check it specially so that
peculiar things can be admitted as pseudo-eof tokens.)

```
   (LET* ((PATH (for SYM in LOOKAHEAD.PATH when (NOT (EQ SYM (QUOTE EOF))) collect SYM))
          (LPLEN (LENGTH LOOKAHEAD.PATH))
          (PLEN (LENGTH PATH)))
         (if (EQ LPLEN PLEN)
             then                                            (* No eof)
                  (if (EQ PLEN 1)
```

```
                    then                                          (* Single-token lookahead)
                         (LIST (QUOTE EQ)
                               (QUOTE CLASS)
                               (KWOTE (CAR PATH)))
                    else                                          (* Multiple-token lookahead)
                         (LIST (QUOTE EQUAL)
                               (QUOTE CLASS)
                               (KWOTE PATH)))
               else                                               (* Check for eof last)
                    (if (EQ LPLEN 1)
                        then                                      (* just check eof)
                             (LIST (fetch EOFFN of PARSERSPEC)
                                   (QUOTE CLASS))
                      else                                        (* Check other tokens first, then eof)
                           (LIST (QUOTE AND)
                                 (LIST (QUOTE for)
                                       (QUOTE READ)
                                       (QUOTE in)
                                       (QUOTE CLASS)
                                       (QUOTE as)
                                       (QUOTE NEED)
                                       (QUOTE in)
                                       (KWOTE PATH)
                                       (QUOTE always)
                                       (LIST (QUOTE EQ)
                                             (QUOTE READ)
                                             (QUOTE NEED)))
                                 (LIST (fetch EOFFN of PARSERSPEC)
                                       (LIST (QUOTE CAR)
                                             (LIST (QUOTE FNTH)
                                                   (QUOTE CLASS)
                                                   (ADD1 PLEN])

)

(DEFINEQ

(PG.CODE.READ
  [LAMBDA (STATE SYMBOL.TABLE PARSERSPEC)                         (* hts: " 3-Apr-86 16:09")

          (* * Generates the code to read a token from the input stream and act on it appropriately.
          Also records the name of this state on the control stack for later use in backup after reducing.)

    (LIST (LIST (QUOTE *)
                (QUOTE Read))
          (PG.CODE.READ.TOKEN PARSERSPEC (for L in (fetch OUT of STATE) when (PG.TERMINALP (fetch SYM of L)
                                                                                           SYMBOL.TABLE)
                                                                collect (fetch SYM of L)))
          (PG.CODE.READ.SWITCH STATE SYMBOL.TABLE PARSERSPEC])

(PG.CODE.READ.TOKEN
  [LAMBDA (PARSERSPEC EXPECTED)                                   (* hts: "28-Feb-86 21:59")

          (* * Generates the code for reading a token from the input stream.
          Reads token from lookahead buffer if anything there, else from stream itself.
          Determines the class and instance of the token. Pushes the instance on the user stack and returns the class.)

    (LIST (QUOTE LET)
          (LIST (LIST (QUOTE EXPECTED.OWN)
                      (KWOTE EXPECTED)))
          [LIST (QUOTE SETQ)
                (QUOTE TOKEN)
                (LIST (QUOTE if)
                      (LIST (fetch QUEUENOTEMPTYFN of PARSERSPEC)
                            (QUOTE LOOKAHEAD))
                      (QUOTE then)
                      (LIST (fetch DEQUEUEFN of PARSERSPEC)
                            (QUOTE LOOKAHEAD))
                      (QUOTE else)
                      (LIST (fetch READFN of PARSERSPEC)
                            (QUOTE EXPECTED.OWN)
                            (LIST (QUOTE CDR)
                                  (QUOTE STATE]
          (LIST (QUOTE SETQ)
                (QUOTE CLASS)
                (LIST (fetch CLASSFN of PARSERSPEC)
                      (QUOTE TOKEN)
                      (QUOTE EXPECTED.OWN)))
          (LIST (fetch PUSHFN of PARSERSPEC)
                (QUOTE USERSTACK)
                (LIST (fetch INSTANCEFN of PARSERSPEC)
                      (QUOTE TOKEN)
                      (QUOTE EXPECTED.OWN)))
          (QUOTE CLASS])
```

## (PG.CODE.READ.SWITCH

```
  [LAMBDA (STATE SYMBOL.TABLE PARSERSPEC)                    (* hts: " 3-Apr-86 16:08")

          (* * Generates the code to make appropriate transitions from lookahead states.
          EOF transitions checked specially (and last), because of possible permissivity in what counts as EOF.
          Note for read transitions, you must record your state path on the control stack so subsequent reduces can know where to
          unwind to.)

    (CONS (QUOTE SELECTQ)
          (CONS (QUOTE CLASS)
                (NCONC1 [for L in (fetch OUT of STATE) when (AND (PG.TERMINALP (fetch SYM of L)
                                                                               SYMBOL.TABLE)
                                                                 (NEQ (QUOTE EOF)
                                                                      (fetch SYM of L)))
                        collect (LIST (fetch SYM of L)
                                      [LIST (fetch PUSHFN of PARSERSPEC)
                                            (QUOTE CONTROLSTACK)
                                            (KWOTE (fetch NAME of (CAR (fetch ST of L]
                                      (LIST (QUOTE GO)
                                            (fetch NAME of (CAR (fetch ST of L]
                              (LET [(EOF.TRANSITION (for L in (fetch OUT of STATE)
                                                       thereis (EQ (QUOTE EOF)
                                                                   (fetch SYM of L]
                                   (if EOF.TRANSITION
                                       then [LET [(NEXT (fetch NAME of (CAR (fetch ST of EOF.TRANSITION]
                                                  (LIST (QUOTE if)
                                                        (LIST (fetch EOFFN of PARSERSPEC)
                                                              (QUOTE CLASS))
                                                        (QUOTE then)
                                                        (LIST (fetch ENQUEUEFN of PARSERSPEC)
                                                              (QUOTE LOOKAHEAD)
                                                              (QUOTE TOKEN))
                                                        (LIST (fetch PUSHFN of PARSERSPEC)
                                                              (QUOTE CONTROLSTACK)
                                                              (KWOTE NEXT))
                                                        (LIST (QUOTE GO)
                                                              NEXT)
                                                        (QUOTE else)
                                                        (LIST (QUOTE PARSE.ERROR]
                                     else (LIST (QUOTE PARSE.ERROR])

)

(DEFINEQ
```

## (PG.CODE.REDUCE

```
  [LAMBDA (STATE PARSERSPEC)                                 (* hts: " 3-Apr-86 16:09")

          (* * Outputs the code for a REDUCE operation.)

    (LET [(PROD (fetch SYM of (for L in (fetch OUT of STATE) thereis (type? PRODUCTION (fetch SYM of L]
                                                                (* PROD is the production rule according to which we will
                                                                reduce.)
         (if (EQ (fetch LHS of PROD)
                 (QUOTE START))
             then

          (* * if you're reducing the start state, you're done: just return the user stack)

                [LIST (LIST (QUOTE *)
                            (QUOTE Reduce)
                            (QUOTE start)
                            (QUOTE state))
                      (LIST (QUOTE RPLACA)
                            (QUOTE STATE)
                            (LIST (fetch SAVESTATEFN of PARSERSPEC)
                                  (QUOTE LOOKAHEAD)))
                      (LIST (fetch POPFN of PARSERSPEC)
                            (QUOTE USERSTACK))
                      (LIST (QUOTE RETURN)
                            (LIST (fetch POPFN of PARSERSPEC)
                                  (QUOTE USERSTACK]
           else

          (* * Otherwise generate code to munge stacks and proceed to the next appropriate state.)

               (LIST (LIST (QUOTE *)
                           (QUOTE Reduce))
                     (PG.CODE.REDUCE.STACKS (fetch LHS of PROD)
                            (fetch RHS of PROD)
                            (fetch (PRODUCTION AUGMENT) of PROD)
                            PARSERSPEC)
                     (PG.CODE.REDUCE.SWITCH STATE (fetch LHS of PROD)
                            (fetch RHS of PROD)
                            PARSERSPEC])
```

## (PG.CODE.REDUCE.STACKS

```
  [LAMBDA (LHS RHS AUGMENT PARSERSPEC)                    (* hts: " 1-Mar-86 15:44")
```

(* * Generates the code to perform a reduce action for the parser.
Must pop the things being reduced off the user stack and replace them with the result of the semantic action for this rule.
Then must pop states being reduced off the control stack. They will later be replaced by the name of the state to which the LHS transition goes.)

```
    (LET ((LEN (LENGTH RHS)))
         (LIST (QUOTE PROGN)
               (LIST (QUOTE SETQ)
                     (QUOTE LHS)
                     (KWOTE LHS))
               (LIST (QUOTE SETQ)
                     (QUOTE RHS)
                     NIL)
               (LIST (QUOTE to)
                     LEN
                     (QUOTE do)
                     (LIST (QUOTE push)
                           (QUOTE RHS)
                           (LIST (fetch POPFN of PARSERSPEC)
                                 (QUOTE USERSTACK)))
                     (LIST (fetch POPFN of PARSERSPEC)
                           (QUOTE CONTROLSTACK)))
               (LIST (fetch PUSHFN of PARSERSPEC)
                     (QUOTE USERSTACK)
                     (COPY AUGMENT))
               (QUOTE LHS])
```

## (PG.CODE.REDUCE.SWITCH

```
  [LAMBDA (S LHS RHS PARSERSPEC)                          (* hts: " 3-Apr-86 16:09")
```

(* * Generates the code to go to the next state from a reduce state.
Where you go depends on where you have been already (your left-context), which can be determined from the top of the control stack.)

```
    (CONS (QUOTE SELECTQ)
          (CONS (LIST (fetch TOPFN of PARSERSPEC)
                      (QUOTE CONTROLSTACK))
                (NCONC1 [for FROM.STATE in (PG.BACKUP (LIST S)
                                                      (REVERSE RHS))
                           collect (LET [(TARGET.NAME (fetch NAME of (PG.REDUCE.TARGET FROM.STATE LHS]
                                                (LIST (fetch NAME of FROM.STATE)
                                                      (LIST (fetch PUSHFN of PARSERSPEC)
                                                            (QUOTE CONTROLSTACK)
                                                            (KWOTE TARGET.NAME))
                                                      (LIST (QUOTE GO)
                                                            TARGET.NAME]
                        (LIST (QUOTE SHOULDNT])

)
```

(* * Other)

```
(DEFINEQ
```

## (PRINT.FUNCTION

```
  [LAMBDA (FUNC FILE)                                     (* hts: "24-Feb-86 22:44")
```

(* * PRETTY-PRINTS FUNCTION FUNC TO FILE (OR THE DEFAULT PRINTER IF FILE IS NIL))

```
    (LET [(DEF (OR (LISTP (GETD FUNC))
                   (LISTP (GETPROP FUNC (QUOTE EXPR]
          (if DEF
              then (LET [(S (OPENSTREAM (OR FILE (QUOTE {LPT}))
                                        (QUOTE OUTPUT]
                        (PRINTDEF (LIST FUNC DEF)
                                  NIL T NIL NIL S)
                        (CLOSEF S])

)

(PUTPROPS PARSER COPYRIGHT ("Xerox Corporation" 1983 1984 1986))
```

## FUNCTION INDEX



## RECORD INDEX



## MACRO INDEX