File created: 10-May-88 14:14:46 **{DSK}<LISPFILES>LOOPS>LYRIC>RULES>LOOPSRULESP.;2**

changes to: (FNS FlushRule LetterP Next?CompoundSeparator ParseAtom ParseBackSlash ParseColon ParseCompoundSeparator ParseDot ParseEqSign ParseGreaterSign ParseLeftArrow ParseLessSign ParseLiteral ParseMinus ParseNotSign ParseNumber ParseOneCharToken ParsePlus ParseString ParseTokens ScanFor SkipRule UnParseTerm)

previous date: 14-Dec-87 21:35:01 {DSK}<LISPFILES>LOOPS>LYRIC>RULES>LOOPSRULESP.;1

Read Table: INTERLISP

Package: INTERLISP

Format: XCCS


```
(RPAQQ LOOPSRULESPCOMS
       [(DECLARE%: DONTCOPY (PROP MAKEFILE-ENVIRONMENT LOOPSRULESP))
                                                                  ; Copyright (c) 1982 by Xerox Corporation
                                                                  ; Written in August 1982 by Mark Stefik, Alan Bell, and Danny
                                                                  ; Bobrow.
                                                                  ; Fns for Parsing RuleSets.
        (FNS * RULEPARSEFNS)
                                                                  ; Vars and constants for RuleSet parsing.
        (E (for VAR in RULEVARS do (SET VAR NIL)))
        (VARS * RULEVARS)
        (GLOBALVARS * RULEVARS)
        (CONSTANTS * RULECONSTANTS)
                                                                  ; Globals for the RuleSet compiler.
        (P (SETQ GLOBALVARS (APPEND GLOBALVARS RULEVARS])

(DECLARE%: DONTCOPY

(PUTPROPS LOOPSRULESP MAKEFILE-ENVIRONMENT (:PACKAGE "IL" :READTABLE "INTERLISP" :BASE 10))
)
```

;; Copyright (c) 1982 by Xerox Corporation

;; Written in August 1982 by Mark Stefik, Alan Bell, and Danny Bobrow.

;; Fns for Parsing RuleSets.

```
(RPAQQ RULEPARSEFNS (FlushRule LetterP Next?CompoundSeparator ParseAtom ParseBackSlash ParseColon
                           ParseCompoundSeparator ParseDot ParseEqSign ParseGreaterSign ParseLeftArrow
                           ParseLessSign ParseLiteral ParseMinus ParseNotSign ParseNumber ParseOneCharToken
                           ParsePlus ParseString ParseTokens ScanFor SkipRule UnParseTerm))

(DEFINEQ
```

(FlushRule

```
  [LAMBDA (errorMsg boldToken moreMsg)                            (* mjs%: "14-FEB-83 12:00")
```

(* * Error Routine activated during RuleSet Parsing. Prints error message and then discards ruleSetTokens remaining in this rule.)

```
    (PROG (token)
          (SETQ parseErrorFlg T)
          (printout NIL T errorMsg)
          (COND
             (boldToken (printout NIL " " .FONT BOLDFONT boldToken .FONT DEFAULTFONT)))
          (COND
             (moreMsg (printout NIL " " moreMsg)))
          (COND
             (ruleSetTokens (printout T " error near: ")
                    (for I from 1 to 5 as token in ruleSetTokens when token do (PRIN1 (UnParseTerm token))
                                                                               (SPACES 1))
                    (TERPRI)))
          (COND
             (rsCompilerDebugFlg (PAUSE "Push RETURN to continue.")))
          (CLEARBUF)
          (SkipRule])
```

(LetterP

```
  [LAMBDA (character)                                             (* mjs%: "16-AUG-82 11:42")
```

(* * Returns T if the character is an alphabetic letter and NIL otherwise.)

```
    (PROG (code flg)
          (SETQ code (CHCON1 character))
          [SETQ flg (OR (AND (IGREATERP code 64)
                             (ILESSP code 91))
                        (AND (IGREATERP code 96)
                             (ILESSP code 123]
          (RETURN flg])
```

(Next?CompoundSeparator

```
[LAMBDA NIL                                                    (* mjs%: " 9-FEB-83 11:22")
```

(* * Subroutine of ParseLiteral. Looks ahead in ruleParseSource to see whether a compound separator follows. Returns T if yes, and NIL otherwise. Does not change ruleParseSource or char.)

```
   (PROG [nextChar (oneCharSeprs (CONSTANT (LIST dot colon comma)))
                  (afterDot (CONSTANT (LIST dot comma bang)))
                  (afterComma (CONSTANT (LIST bang)))
                  (afterColon (CONSTANT (LIST colon comma bang]
         (SETQ nextChar (SUBATOM ruleParseSource 1 1))
         (RETURN (OR (AND (FMEMB ruleParseChar oneCharSeprs)
                          (OR (LetterP nextChar)
                              (EQ nextChar upArrow)))
                     (AND (EQ ruleParseChar dot)
                          (FMEMB nextChar afterDot))
                     (AND (EQ ruleParseChar colon)
                          (FMEMB nextChar afterColon))
                     (AND (EQ ruleParseChar comma)
                          (FMEMB nextChar afterComma])
```

(ParseAtom

```
[LAMBDA NIL                                                    (* mjs%: " 9-FEB-83 12:07")
```

(* * Subroutine of ParseLiteral. Recognizes atoms. Input string is in global variable ruleSetSource. Returns parsed atom.)

```
   (PACK (CONS ruleParseChar (while (AND (SETQ ruleParseChar (GNC ruleParseSource))
                                         (OR (LetterP ruleParseChar)
                                             (NUMBERP ruleParseChar)))
                               collect ruleParseChar])
```

(ParseBackSlash

```
[LAMBDA NIL                                                    (* mjs%: " 8-JUN-83 10:41")
```

(* * Subroutine of ParseTokens. Recognizes ruleSetTokens starting with a backSlash. Input string is in global variable ruleSetSource. Returns parsed token.)

```
   (PROG (nextChar)
         (SETQ nextChar (SUBATOM ruleParseSource 1 1))
         (RETURN (COND
                    ((LetterP nextChar)                        (* Here for / lisp variables.)
                     (SETQ ruleParseChar (GNC ruleParseSource))
                     (ParseLiteral 'LispVar backSlash))
                    (T (SETQ parseErrorFlg nextChar)
                       (SETQ ruleParseChar (GNC ruleParseSource])
```

(ParseColon

```
[LAMBDA NIL                                                    (* mjs%: "11-FEB-83 14:37")
```

(* * Subroutine of ParseTokens. Recognizes ruleSetTokens starting with a colon. Input string is in global variable ruleSetSource. Returns parsed token.)

```
   (PROG (nextChar)
         (SETQ nextChar (SUBATOM ruleParseSource 1 1))
         (RETURN (COND
                    ((AND (EQ bang nextChar)
                          (LetterP (SUBSTRING ruleParseSource 2 2)))
                                                               (* here for %:! to self.)
                     (SETQ ruleParseChar (GNC ruleParseSource))
                     (SETQ ruleParseChar (GNC ruleParseSource))
                     (ParseLiteral 'self colonBang))
                    ((LetterP nextChar)                        (* Here for %: to self.)
                     (SETQ ruleParseChar (GNC ruleParseSource))
                     (ParseLiteral 'self colon))
                    [(EQ colon nextChar)                       (* Here for |::| constructs.)
                     (SETQ ruleParseChar (GNC ruleParseSource))
                     (SETQ nextChar (SUBATOM ruleParseSource 1 1))
                     (COND
                        ((LetterP nextChar)                    (* here for |::| to self.)
                         (SETQ ruleParseChar (GNC ruleParseSource))
                         (ParseLiteral 'self coloncolon))
                        ((AND (EQ bang nextChar)
                              (LetterP (SUBSTRING ruleParseSource 2 2)))
                                                               (* here for |::!| to self.)
                         (SETQ ruleParseChar (GNC ruleParseSource))
                         (SETQ ruleParseChar (GNC ruleParseSource))
                         (ParseLiteral 'self colonColonBang))
                        (T (SETQ parseErrorFlg nextChar)
                           (SETQ ruleParseChar (GNC ruleParseSource]
                    (T                                         (* Here for %: all by itself.)
                       (SETQ ruleParseChar (GNC ruleParseSource))
```

```
                        colon])
```

(ParseCompoundSeparator

```
  [LAMBDA NIL                                                    ; Edited 14-Dec-87 21:34 by jrb:
```

(* * Subroutine of ParseLiteral. Recognizes compoundSeparators.
Input string is in global variable ruleSetSource. Returns parsed separator.)

```
    (PROG (separator)
          (SETQ separator ruleParseChar)                         (* Look ahead one character to check for dotdot, coloncolon etc.)
          (SETQ ruleParseChar (GNC ruleParseSource))
          (RETURN (COND
                     ((OR (LetterP ruleParseChar)
                          (EQ ruleParseChar upArrow))            (* Here for %: or %. or (\, compound.))
                      separator)
                     ((AND (EQ separator colon)
                           (EQ ruleParseChar bang))              (* Here for %:! compound.)
                      (SETQ ruleParseChar (GNC ruleParseSource))
                      colonBang)
                     ((AND (EQ separator comma)
                           (EQ ruleParseChar bang))              (* Here for %,! compound.)
                      (SETQ ruleParseChar (GNC ruleParseSource))
                      commaBang)
                     ((AND (EQ separator colon)
                           (EQ ruleParseChar colon))
                      (SETQ ruleParseChar (GNC ruleParseSource))
                      (COND
                         ((EQ ruleParseChar bang)                (* Here for |::!| compound.)
                          (SETQ ruleParseChar (GNC ruleParseSource))
                          colonColonBang)
                         (T                                      (* Here for |::| compound.)
                            coloncolon)))
                     ((AND (EQ separator colon)
                           (EQ ruleParseChar comma))
                      (SETQ ruleParseChar (GNC ruleParseSource))
                      (COND
                         ((EQ ruleParseChar bang)                (* Here for %:,! compound.)
                          (SETQ ruleParseChar (GNC ruleParseSource))
                          colonCommaBang)
                         (T                                      (* Here for %:, compound.)
                            colonComma)))
                     ((EQ separator dot)
                      (COND
                         ((EQ ruleParseChar bang)                (* Here for .! compound.)
                          (SETQ ruleParseChar (GNC ruleParseSource))
                          dotBang)
                         ((EQ ruleParseChar dot)
                          (SETQ ruleParseChar (GNC ruleParseSource))
                          (COND
                             ((EQ ruleParseChar star)
                              (SETQ ruleParseChar (GNC ruleParseSource))
                                                                 (* Here for ..* compound.)
                              dotDotStar)
                             (T                                  (* Here for |..| compound.)
                                dotdot)))
                         ((EQ ruleParseChar comma)               (* Here for ., compound.)
                          (SETQ ruleParseChar (GNC ruleParseSource))
                          dotcomma)))
                     (T                                          (* Invalid compound separator.)
                        (SETQ parseErrorFlg ruleParseChar])
```

(ParseDot

```
  [LAMBDA NIL                                                    (* mjs%: "11-FEB-83 11:48")
```

(* * Subroutine of ParseTokens. Recognizes ruleSetTokens starting with a period.
Input string is in global variable ruleSetSource. Returns parsed token.)

```
    (PROG (nextChar)
          (SETQ nextChar (SUBATOM ruleParseSource 1 1))
          (RETURN (COND
                     ((NUMBERP nextChar)                         (* Here for floating point numbers.)
                      (ParseNumber))
                     ((AND (EQ bang nextChar)
                           (LetterP (SUBSTRING ruleParseSource 2 2)))
                                                                 (* here for .! msgs to self.)
                      (SETQ ruleParseChar (GNC ruleParseSource))
                      (SETQ ruleParseChar (GNC ruleParseSource))
                      (ParseLiteral 'self dotBang))
                     ((LetterP nextChar)                         (* Here for %. messages to self.)
                      (SETQ ruleParseChar (GNC ruleParseSource))
                      (ParseLiteral 'self dot))
                     (T (SETQ parseErrorFlg nextChar)
                        (SETQ ruleParseChar (GNC ruleParseSource])
```

(ParseEqSign

```
[LAMBDA NIL                                                    (* mjs%: "22-JAN-83 09:26")
```

(* * Subroutine of ParseTokens. Recognizes ruleSetTokens starting with eqSign --
either = or ==. Input string is in global variable ruleSetSource. Returns parsed token.)

```
   (SETQ ruleParseChar (GNC ruleParseSource))
   (COND
      ((EQ ruleParseChar eqSign)                               (* Here for ==)
       (SETQ ruleParseChar (GNC ruleParseSource))
       eqeqSign)
      (T                                                       (* Here for =)
         eqSign])
```

(ParseGreaterSign

```
[LAMBDA NIL                                                    (* mjs%: "22-JAN-83 09:26")
```

(* * Subroutine of ParseTokens. Recognizes ruleSetTokens starting with greaterSign either > or >=.
Input string is in global variable ruleSetSource. Returns parsed token.)

```
   (SETQ ruleParseChar (GNC ruleParseSource))
   (COND
      ((EQ ruleParseChar eqSign)                               (* Here for >=)
       (SETQ ruleParseChar (GNC ruleParseSource))
       greaterEqSign)
      (T                                                       (* Here for >)
         greaterSign])
```

(ParseLeftArrow

```
[LAMBDA NIL                                                    (* mjs%: "22-JAN-83 09:26")
```

(* * Subroutine of ParseTokens. Recognizes ruleSetTokens starting with leftArrow --
either _ or _+ or _-. Input string is in global variable ruleSetSource.
Returns parsed token.)

```
   (SETQ ruleParseChar (GNC ruleParseSource))
   (COND
      ((EQ ruleParseChar plus)                                 (* Here for _+)
       (SETQ ruleParseChar (GNC ruleParseSource))
       push)
      ((EQ ruleParseChar minus)                                (* Here for _-)
       (SETQ ruleParseChar (GNC ruleParseSource))
       pop)
      ((EQ ruleParseChar bang)                                 (* Here for _!)
       (SETQ ruleParseChar (GNC ruleParseSource))
       leftArrowBang)
      (T                                                       (* Here for _)
         leftArrow])
```

(ParseLessSign

```
[LAMBDA NIL                                                    (* mjs%: "22-JAN-83 09:27")
```

(* * Subroutine of ParseTokens. Recognizes ruleSetTokens starting with lessSign either < or <=.
Input string is in global variable ruleSetSource. Returns parsed token.)

```
   (SETQ ruleParseChar (GNC ruleParseSource))
   (COND
      ((EQ ruleParseChar eqSign)                               (* Here for <=)
       (SETQ ruleParseChar (GNC ruleParseSource))
       lessEqSign)
      ((EQ ruleParseChar lessSign)                             (* Here for <<)
       (SETQ ruleParseChar (GNC ruleParseSource))
       membSign)
      (T                                                       (* Here for <)
         lessSign])
```

(ParseLiteral

```
[LAMBDA (leftLiteral separator)                                (* mjs%: " 8-JUN-83 10:45")
```

(* * Subroutine of ParseTokens. Recognizes literals. Input string is in global variable ruleSetSource.
Returns parsed literal. If literal is compound, returns a list of the separator followed by the two parts of the compound.
For example, RS..ws parses to (|..| RS ws)%. ParseLiteral recurs to handle nested compound literals, such as obj.sel%:var
which parses to (%: (%. obj sel) var)%. On recursive calls, the argument leftLiteral contains the literal that is the left part.)

```
   (PROG (atom dollarFlg)                                      (* Parse the next atom. Special treatment if preceded by dollar sign.)
         [COND
            ((EQ ruleParseChar dollarSign)
             (SETQ dollarFlg T)
             (SETQ ruleParseChar (GNC ruleParseSource]
         (SETQ atom (ParseAtom))
         [COND
            (dollarFlg (SETQ atom (LIST dollarSign atom]
```

```
          [COND
             ((NULL leftLiteral)                                    (* Here for Non-Recursive Call)
              (SETQ leftLiteral atom)
              (COND
                 ((Next?CompoundSeparator)                          (* if compound, set up as recursive call and fall thru.)
                  (SETQ separator (ParseCompoundSeparator))
                  (SETQ atom (ParseAtom)))
                 ((EQ ruleParseChar dot)                            (* Here to handle non-compound use of period as in comments.)
                  (SETQ ruleParseChar (GNC ruleParseSource))
                  (SETQ atom (MKATOM (CONCAT atom dot)))
                  (RETURN atom))
                 (T                                                 (* if simple, then just return atom.)
                    (RETURN atom]
```

(* * Here for Recursive Call. leftLiteral and separator are set.)

```
          (RETURN (COND
                     ((Next?CompoundSeparator)                      (* Here to recur again.)
                      (SETQ leftLiteral (LIST separator leftLiteral atom))
                      (SETQ separator (ParseCompoundSeparator))
                      (ParseLiteral leftLiteral separator))
                     (T                                             (* Here if last Separator.)
                        (LIST separator leftLiteral atom])
```

(ParseMinus

```
  [LAMBDA NIL                                                       (* mjs%: " 3-MAR-83 16:38")
```

(* * Subroutine of ParseTokens. Recognizes ruleSetTokens starting with plus --
either -
or ->. Also detects use of unary minus. Input string is in global variable ruleSetSource.
Returns parsed token.)

```
    (PROG [(unarySignals (CONSTANT (LIST eqSign eqeqSign lessSign greaterSign lpar leftArrow rightArrow
                                   semicolon colon plus minus asterisk slash]
          (SETQ ruleParseChar (GNC ruleParseSource))
          (RETURN (COND
                     ((EQ ruleParseChar greaterSign)                (* Here for ->)
                      (SETQ ruleParseChar (GNC ruleParseSource))
                      rightArrow)
                     ((EQ ruleParseChar minus)                      (* Here for --)
                      (SETQ ruleParseChar (GNC ruleParseSource))
                      minusminus)
                     ([OR (NULL ruleSetTokens)
                          (FMEMB (CAR ruleSetTokens)
                                 unarySignals)
                          (FMEMB (CAR ruleSetTokens)
                                 thenSpellings)
                          (EQ lpar (CADR ruleSetTokens))
                          (AND (EQ lbracket (CADDDR ruleSetTokens))
                               (EQ leftArrow (CADDR ruleSetTokens]
                                                                    (* Here for -1-)
                                                                    (* cases include msgs and fn calls.)
                      unaryMinus)
                     (T                                             (* Here for -)
                        minus])
```

(ParseNotSign

```
  [LAMBDA NIL                                                       (* mjs%: "22-JAN-83 09:27")
```

(* * Subroutine of ParseTokens. Recognizes ruleSetTokens starting with notSign.
Input string is in global variable ruleSetSource. Returns parsed token.)

```
    (SETQ ruleParseChar (GNC ruleParseSource))
    (COND
       ((EQ ruleParseChar eqSign)                                   (* Here for ~=)
        (SETQ ruleParseChar (GNC ruleParseSource))
        neqSign)
       (T                                                           (* Here for ~)
          notSign])
```

(ParseNumber

```
  [LAMBDA NIL                                                       (* mjs%: "22-JAN-83 09:27")
```

(* * Subroutine of ParseTokens. Recognizes numbers. Input string is in global variable ruleSetSource.
Returns parsed number.)

```
    (PROG (chars token)
      CollectChars
          (SETQ chars (CONS ruleParseChar chars))
      NextChar
          (SETQ ruleParseChar (GNC ruleParseSource))
          (COND
             ((OR (NUMBERP ruleParseChar)
                  (EQ ruleParseChar period))                        (* Collect numbers and periods for floating point numbers.)
```

```
            (GO CollectChars)))

        (* * Check that token is valid number and return to ParseTokens.)

        (SETQ token (PACK (DREVERSE chars)))
        (COND
           ((NOT (NUMBERP token))                          (* Set errorFlg if not valid number.)
            (SETQ parseErrorFlg token)))
        (RETURN token])
```

(ParseOneCharToken

```
  [LAMBDA NIL                                              (* mjs%: " 9-FEB-83 14:25")
```

(* * Subroutine of ParseTokens. Recognizes ruleSetTokens consisting of a single character.
Input string is in global variable ruleSetSource. Returns parsed reserved word.)

```
    (PROG1 ruleParseChar
        (SETQ ruleParseChar (GNC ruleParseSource)))])
```

(ParsePlus

```
  [LAMBDA NIL                                              (* mjs%: "22-JAN-83 09:27")
```

(* * Subroutine of ParseTokens. Recognizes ruleSetTokens starting with plus --
either + or ++. Input string is in global variable ruleSetSource. Returns parsed token.)

```
    (SETQ ruleParseChar (GNC ruleParseSource))
    (COND
       ((EQ ruleParseChar plus)                            (* Here for ++)
        (SETQ ruleParseChar (GNC ruleParseSource))
        plusplus)
       (T                                                  (* Here for +)
          plus])
```

(ParseString

```
  [LAMBDA NIL                                              ; Edited 10-May-88 13:53 by hwl
```

(* * Subroutine of ParseTokens. Recognizes strings. Input string is in global variable ruleSetSource.
Returns parsed string.)

```
    (PROG (chars)                                          (* Skip the leading stringSign)
          (SETQ ruleParseChar (GNC ruleParseSource))
          [SETQ chars (while (AND ruleParseChar (NEQ ruleParseChar semicolon)
                                  (NEQ ruleParseChar stringSign))
                         collect (PROG1 ruleParseChar
                                        (SETQ ruleParseChar (GNC ruleParseSource)))]
          [COND
             ((EQ ruleParseChar stringSign)
              (SETQ ruleParseChar (GNC ruleParseSource)))
             (T                                            (* Error if scan did not end on a stringSign)
                (SETQ parseErrorFlg ruleParseChar)
                (RS.WRITE "Missing end of string: " (MKSTRING chars]
          (RETURN (MKSTRING (PACK chars])
```

(ParseTokens

```
  [LAMBDA (sourceRules)                                    ; Edited 10-May-88 13:54 by hwl
```

(* * First pass of RuleSet compilation by parsing the ruleParseSource string into a list of tokens.
Operates as a state machine for recognizing the tokens. During parsing uses global variables ruleParseChar and
ruleSetSource. Value returned in the global variable tokens.)

```
    (PROG [token everErrorFlg
           (oneCharTokens (CONSTANT (LIST asterisk slash quoteSign lpar rpar semicolon lbracket rbracket
                                          bang lbrace rbrace comma questionmark upArrow)))
           (skipChars (CONSTANT (LIST carriageReturn lineFeed space tab]
          (SETQ parseErrorFlg NIL)
          (SETQ ruleParseSource (COPYALL sourceRules))
          (SETQ ruleSetTokens NIL)
      NextChar
          (SETQ ruleParseChar (GNC ruleParseSource))
```

(* * Each State is a subroutine using global variables ruleParseChar and ruleSetSource.
Each subroutine removes characters from ruleParseSource until its token is complete and returns the token as its value.
Since the subroutine takes one more character than it needs, the global variable ruleParseChar is set to the first character
for the next state.)

```
      NextState
          [SETQ token (COND
                         ((NULL ruleParseChar)
                          (GO Done))
                         ((FMEMB ruleParseChar skipChars)
                          (GO NextChar))
                         ((NUMBERP ruleParseChar)
                          (ParseNumber))
```

```
                        ((OR (LetterP ruleParseChar)
                             (EQ ruleParseChar dollarSign))
                         (ParseLiteral))
                        ((EQ ruleParseChar dot)
                         (ParseDot))
                        ((EQ ruleParseChar colon)
                         (ParseColon))
                        ((EQ ruleParseChar backSlash)
                         (ParseBackSlash))
                        ((FMEMB ruleParseChar oneCharTokens)
                         (ParseOneCharToken))
                        ((EQ ruleParseChar lessSign)
                         (ParseLessSign))
                        ((EQ ruleParseChar greaterSign)
                         (ParseGreaterSign))
                        ((EQ ruleParseChar leftArrow)
                         (ParseLeftArrow))
                        ((EQ ruleParseChar eqSign)
                         (ParseEqSign))
                        ((EQ ruleParseChar plus)
                         (ParsePlus))
                        ((EQ ruleParseChar minus)
                         (ParseMinus))
                        ((EQ ruleParseChar notSign)
                         (ParseNotSign))
                        ((EQ ruleParseChar stringSign)
                         (ParseString))
                        (T (RS.WRITE "Unexpected " ruleParseChar " in " (CADR ruleSetTokens)
                                  (CAR ruleSetTokens)
                                  ruleParseChar
                                  (SUBSTRING ruleParseSource 1 10))
                           (SETQ everErrorFlg T)
                           (GO NextChar]

          (* * Here on return from state.)

          [COND
             (parseErrorFlg (RS.WRITE "Unexpected " parseErrorFlg " in " (CADR ruleSetTokens)
                                  (CAR ruleSetTokens)
                                  parseErrorFlg
                                  (SUBSTRING ruleParseSource 1 10))
                    (SETQ parseErrorFlg NIL)
                    (SETQ everErrorFlg T))
             (T                                                    (* Normal state return. Save token.)
                (SETQ ruleSetTokens (CONS token ruleSetTokens]
          (GO NextState)
      Done

          (* * Here when done with Source. Add extra semicolon to end and reverse the list.)

          (SETQ ruleSetTokens (CONS semicolon ruleSetTokens))
          (SETQ ruleSetTokens (DREVERSE ruleSetTokens))
          (SETQ parseErrorFlg everErrorFlg)
          (RETURN parseErrorFlg])
```

(ScanFor

```
  [LAMBDA (goodList stopList)                                      (* mjs%: "21-JAN-83 14:29")
```

(* Scans global variable Tokens looking for next occurrence of one of the ruleSetTokens on goodList. Returns the first one found or NIL if none were found.)

```
    (for token in ruleSetTokens until (FMEMB token stopList) thereis (FMEMB token goodList])
```

(SkipRule

```
  [LAMBDA NIL                                                      (* mjs%: " 8-JUN-83 11:29")
```

(* * Pop the ruleSetTokens for the current rule through a semicolon.)

```
    (PROG (token)
          (do (SETQ token (pop ruleSetTokens)) repeatuntil (OR (EQ token semicolon)
                                                               (NULL ruleSetTokens])
```

(UnParseTerm

```
  [LAMBDA (parsedTerm)                                             (* dgb%: "21-Feb-84 10:23")
```

(* * Returns a term in the ruleParseSource Rule Language given its parsed form. Value is a string.)

```
    (COND
       ((LITATOM parsedTerm)
        parsedTerm)
       ((NUMBERP parsedTerm)
        parsedTerm)
       [(LISTP parsedTerm)
```

```
        (COND
           ((EQ (CAR parsedTerm)
                '\)                                          (* Special case for LispVars)
            (CONCAT '\ (CADDR parsedTerm)))
           (T (CONCAT (UnParseTerm (CADR parsedTerm))
                      (CAR parsedTerm)
                      (CADDR parsedTerm]
       (T (PAUSE "Bad Call to UnParseTerm."])

)
```

;; Vars and constants for RuleSet parsing.

```
(RPAQQ RULEVARS
       (auditSpecification controlType debugVars oneShotBangFlg oneShotFlg parseErrorFlg reEditMenu rsArgs
              rsAuditClass rsAuditFlg rsAuditSpecification rsBreakFlg rsCompilerDebugFlg rsCompilerOptions
              rsInternalTaskVars rsInternalTempVars rsLispCompileFlg rsName rsNumRules rsPrintRuleFlg
              rsRuleAppliedFlg rsRuleClass rsRuleObjects rsSomeDeclChanged rsSomeRuleAuditFlg rsTaskFlg
              rsTraceFlg rsWhileCondition ruleAuditFlg ruleAuditSpecification ruleBreakFlg ruleLabel
              ruleMakeAuditRecordFlg ruleMetaTokens ruleNeedsAuditFlg ruleNumber ruleObject ruleParseChar
              ruleParseSource ruleRHSFlg ruleSetTokens ruleTraceFlg ruleVars taskVars tempVars wsClass wsVars))

(RPAQQ auditSpecification NIL)

(RPAQQ controlType NIL)

(RPAQQ debugVars NIL)

(RPAQQ oneShotBangFlg NIL)

(RPAQQ oneShotFlg NIL)

(RPAQQ parseErrorFlg NIL)

(RPAQQ reEditMenu NIL)

(RPAQQ rsArgs NIL)

(RPAQQ rsAuditClass NIL)

(RPAQQ rsAuditFlg NIL)

(RPAQQ rsAuditSpecification NIL)

(RPAQQ rsBreakFlg NIL)

(RPAQQ rsCompilerDebugFlg NIL)

(RPAQQ rsCompilerOptions NIL)

(RPAQQ rsInternalTaskVars NIL)

(RPAQQ rsInternalTempVars NIL)

(RPAQQ rsLispCompileFlg NIL)

(RPAQQ rsName NIL)

(RPAQQ rsNumRules NIL)

(RPAQQ rsPrintRuleFlg NIL)

(RPAQQ rsRuleAppliedFlg NIL)

(RPAQQ rsRuleClass NIL)

(RPAQQ rsRuleObjects NIL)

(RPAQQ rsSomeDeclChanged NIL)

(RPAQQ rsSomeRuleAuditFlg NIL)

(RPAQQ rsTaskFlg NIL)

(RPAQQ rsTraceFlg NIL)

(RPAQQ rsWhileCondition NIL)

(RPAQQ ruleAuditFlg NIL)

(RPAQQ ruleAuditSpecification NIL)

(RPAQQ ruleBreakFlg NIL)

(RPAQQ ruleLabel NIL)

(RPAQQ ruleMakeAuditRecordFlg NIL)
```

```
(RPAQQ ruleMetaTokens NIL)

(RPAQQ ruleNeedsAuditFlg NIL)

(RPAQQ ruleNumber NIL)

(RPAQQ ruleObject NIL)

(RPAQQ ruleParseChar NIL)

(RPAQQ ruleParseSource NIL)

(RPAQQ ruleRHSFlg NIL)

(RPAQQ ruleSetTokens NIL)

(RPAQQ ruleTraceFlg NIL)

(RPAQQ ruleVars NIL)

(RPAQQ taskVars NIL)

(RPAQQ tempVars NIL)

(RPAQQ wsClass NIL)

(RPAQQ wsVars NIL)

(RPAQQ RULEVARS
       (auditSpecification controlType debugVars oneShotBangFlg oneShotFlg parseErrorFlg reEditMenu rsArgs
              rsAuditClass rsAuditFlg rsAuditSpecification rsBreakFlg rsCompilerDebugFlg rsCompilerOptions
              rsInternalTaskVars rsInternalTempVars rsLispCompileFlg rsName rsNumRules rsPrintRuleFlg
              rsRuleAppliedFlg rsRuleClass rsRuleObjects rsSomeDeclChanged rsSomeRuleAuditFlg rsTaskFlg
              rsTraceFlg rsWhileCondition ruleAuditFlg ruleAuditSpecification ruleBreakFlg ruleLabel
              ruleMakeAuditRecordFlg ruleMetaTokens ruleNeedsAuditFlg ruleNumber ruleObject ruleParseChar
              ruleParseSource ruleRHSFlg ruleSetTokens ruleTraceFlg ruleVars taskVars tempVars wsClass wsVars))

(DECLARE%: DOEVAL@COMPILE DONTCOPY

(GLOBALVARS auditSpecification controlType debugVars oneShotBangFlg oneShotFlg parseErrorFlg reEditMenu rsArgs
       rsAuditClass rsAuditFlg rsAuditSpecification rsBreakFlg rsCompilerDebugFlg rsCompilerOptions
       rsInternalTaskVars rsInternalTempVars rsLispCompileFlg rsName rsNumRules rsPrintRuleFlg rsRuleAppliedFlg
       rsRuleClass rsRuleObjects rsSomeDeclChanged rsSomeRuleAuditFlg rsTaskFlg rsTraceFlg rsWhileCondition
       ruleAuditFlg ruleAuditSpecification ruleBreakFlg ruleLabel ruleMakeAuditRecordFlg ruleMetaTokens
       ruleNeedsAuditFlg ruleNumber ruleObject ruleParseChar ruleParseSource ruleRHSFlg ruleSetTokens
       ruleTraceFlg ruleVars taskVars tempVars wsClass wsVars)
)

(RPAQQ RULECONSTANTS
       [(asterisk (MKATOM "*"))
        (star (MKATOM "*"))
        (slash (MKATOM "/"))
        (backSlash (MKATOM "\"))
        (lessSign (MKATOM "<"))
        (lessEqSign (MKATOM "<="))
        (greaterSign (MKATOM ">"))
        (greaterEqSign (MKATOM ">="))
        (eqSign (MKATOM "="))
        (eqeqSign (MKATOM "=="))
        (membSign (MKATOM "<<"))
        (notSign (MKATOM "~"))
        (neqSign (MKATOM "~="))
        (quoteSign (MKATOM "'"))
        (stringSign (MKATOM '%"))
        (lpar (MKATOM "("))
        (rpar (MKATOM ")"))
        (upArrow (MKATOM "^"))
        (rightArrow (MKATOM "->"))
        (leftArrow (MKATOM "_"))
        (leftArrowBang (MKATOM "_!"))
        (period (MKATOM "."))
        (dot period)
        (dotBang (MKATOM ".!"))
        (dotdot (MKATOM ".."))
        (dotcomma (MKATOM ".,"))
        (dotDotStar (MKATOM "..*"))
        (colon (MKATOM ":"))
        (coloncolon (MKATOM "::"))
        (colonBang (MKATOM ":!"))
        (colonColonBang (MKATOM "::!"))
        (colonComma (MKATOM ":,"))
        (colonCommaBang (MKATOM ":,!"))
        (semicolon (MKATOM ";"))
        (lbrace (MKATOM "{"))
        (rbrace (MKATOM "}"))
        (lbracket (MKATOM "["))
```

```
        (rbracket (MKATOM "]"))
        (comma (MKATOM ","))
        (commaBang (MKATOM ",!"))
        (questionmark (MKATOM "?"))
        (bang (MKATOM "!"))
        (oneBang (MKATOM "1!"))
        (verticalbar (MKATOM "|"))
        (atsign (MKATOM "@"))
        (sharp (MKATOM "#"))
        (dollarSign (MKATOM "$"))
        (ampersand (MKATOM "&"))
        (carriageReturn (CHARACTER 13))
        (lineFeed (CHARACTER 10))
        (crlf "
              ")
        (space (CHARACTER 32))
        (tab (CHARACTER 9))
        (push (MKATOM "_+"))
        (pop (MKATOM "_-"))
        (minus (MKATOM "-"))
        (unaryMinus (MKATOM "-1-"))
        (plus (MKATOM "+"))
        (plusplus (MKATOM "++"))
        (minusminus (MKATOM "--"))
        (endExpr 'endExpr)
        (^noRuleApplied 'NoRuleApplied)
        (compileTimeVars '(ruleNumber ruleLabel ruleObject))
        (lispConstants (LIST NIL T))
        (cyclicControlStructures '(WHILE1 WHILEALL WHILENEXT FOR1 FORALL))
        (reservedRuleWords '(self ruleApplied))
        (sendSpellings '(_! _ SEND Send send))
        (stopSpellings '(STOP Stop stop))
        (thenSpellings '(THEN Then then ->))
        (ifSpellings '(IF If if])

(DECLARE%: EVAL@COMPILE

(RPAQ asterisk (MKATOM "*"))

(RPAQ star (MKATOM "*"))

(RPAQ slash (MKATOM "/"))

(RPAQ backSlash (MKATOM "\"))

(RPAQ lessSign (MKATOM "<"))

(RPAQ lessEqSign (MKATOM "<="))

(RPAQ greaterSign (MKATOM ">"))

(RPAQ greaterEqSign (MKATOM ">="))

(RPAQ eqSign (MKATOM "="))

(RPAQ eqeqSign (MKATOM "=="))

(RPAQ membSign (MKATOM "<<"))

(RPAQ notSign (MKATOM "~"))

(RPAQ neqSign (MKATOM "~="))

(RPAQ quoteSign (MKATOM "'"))

(RPAQ stringSign (MKATOM '%"))

(RPAQ lpar (MKATOM "("))

(RPAQ rpar (MKATOM ")"))

(RPAQ upArrow (MKATOM "^"))

(RPAQ rightArrow (MKATOM "->"))

(RPAQ leftArrow (MKATOM "_"))

(RPAQ leftArrowBang (MKATOM "_!"))

(RPAQ period (MKATOM "."))

(RPAQ dot period)

(RPAQ dotBang (MKATOM ".!"))

(RPAQ dotdot (MKATOM ".."))
```

```
(RPAQ dotcomma (MKATOM ".,"))

(RPAQ dotDotStar (MKATOM "..*"))

(RPAQ colon (MKATOM ":"))

(RPAQ coloncolon (MKATOM "::"))

(RPAQ colonBang (MKATOM ":!"))

(RPAQ colonColonBang (MKATOM "::!"))

(RPAQ colonComma (MKATOM ":,"))

(RPAQ colonCommaBang (MKATOM ":,!"))

(RPAQ semicolon (MKATOM ";"))

(RPAQ lbrace (MKATOM "{"))

(RPAQ rbrace (MKATOM "}"))

(RPAQ lbracket (MKATOM "["))

(RPAQ rbracket (MKATOM "]"))

(RPAQ comma (MKATOM ","))

(RPAQ commaBang (MKATOM ",!"))

(RPAQ questionmark (MKATOM "?"))

(RPAQ bang (MKATOM "!"))

(RPAQ oneBang (MKATOM "1!"))

(RPAQ verticalbar (MKATOM "|"))

(RPAQ atsign (MKATOM "@"))

(RPAQ sharp (MKATOM "#"))

(RPAQ dollarSign (MKATOM "$"))

(RPAQ ampersand (MKATOM "&"))

(RPAQ carriageReturn (CHARACTER 13))

(RPAQ lineFeed (CHARACTER 10))

(RPAQ crlf "
")

(RPAQ space (CHARACTER 32))

(RPAQ tab (CHARACTER 9))

(RPAQ push (MKATOM "_+"))

(RPAQ pop (MKATOM "_-"))

(RPAQ minus (MKATOM "-"))

(RPAQ unaryMinus (MKATOM "-1-"))

(RPAQ plus (MKATOM "+"))

(RPAQ plusplus (MKATOM "++"))

(RPAQ minusminus (MKATOM "--"))

(RPAQQ endExpr endExpr)

(RPAQQ ^noRuleApplied NoRuleApplied)

(RPAQQ compileTimeVars (ruleNumber ruleLabel ruleObject))

(RPAQ lispConstants (LIST NIL T))

(RPAQQ cyclicControlStructures (WHILE1 WHILEALL WHILENEXT FOR1 FORALL))

(RPAQQ reservedRuleWords (self ruleApplied))

(RPAQQ sendSpellings (_! _ SEND Send send))

(RPAQQ stopSpellings (STOP Stop stop))
```

```
(RPAQQ thenSpellings (THEN Then then ->))

(RPAQQ ifSpellings (IF If if))

[CONSTANTS (asterisk (MKATOM "*"))
       (star (MKATOM "*"))
       (slash (MKATOM "/"))
       (backSlash (MKATOM "\"))
       (lessSign (MKATOM "<"))
       (lessEqSign (MKATOM "<="))
       (greaterSign (MKATOM ">"))
       (greaterEqSign (MKATOM ">="))
       (eqSign (MKATOM "="))
       (eqeqSign (MKATOM "=="))
       (membSign (MKATOM "<<"))
       (notSign (MKATOM "~"))
       (neqSign (MKATOM "~="))
       (quoteSign (MKATOM "'"))
       (stringSign (MKATOM '%"))
       (lpar (MKATOM "("))
       (rpar (MKATOM ")"))
       (upArrow (MKATOM "^"))
       (rightArrow (MKATOM "->"))
       (leftArrow (MKATOM "_"))
       (leftArrowBang (MKATOM "_!"))
       (period (MKATOM "."))
       (dot period)
       (dotBang (MKATOM ".!"))
       (dotdot (MKATOM ".."))
       (dotcomma (MKATOM ".,"))
       (dotDotStar (MKATOM "..*"))
       (colon (MKATOM ":"))
       (coloncolon (MKATOM "::"))
       (colonBang (MKATOM ":!"))
       (colonColonBang (MKATOM "::!"))
       (colonComma (MKATOM ":,"))
       (colonCommaBang (MKATOM ":,!"))
       (semicolon (MKATOM ";"))
       (lbrace (MKATOM "{"))
       (rbrace (MKATOM "}"))
       (lbracket (MKATOM "["))
       (rbracket (MKATOM "]"))
       (comma (MKATOM ","))
       (commaBang (MKATOM ",!"))
       (questionmark (MKATOM "?"))
       (bang (MKATOM "!"))
       (oneBang (MKATOM "1!"))
       (verticalbar (MKATOM "|"))
       (atsign (MKATOM "@"))
       (sharp (MKATOM "#"))
       (dollarSign (MKATOM "$"))
       (ampersand (MKATOM "&"))
       (carriageReturn (CHARACTER 13))
       (lineFeed (CHARACTER 10))
       (crlf "
             ")
       (space (CHARACTER 32))
       (tab (CHARACTER 9))
       (push (MKATOM "_+"))
       (pop (MKATOM "_-"))
       (minus (MKATOM "-"))
       (unaryMinus (MKATOM "-1-"))
       (plus (MKATOM "+"))
       (plusplus (MKATOM "++"))
       (minusminus (MKATOM "--"))
       (endExpr 'endExpr)
       (^noRuleApplied 'NoRuleApplied)
       (compileTimeVars '(ruleNumber ruleLabel ruleObject))
       (lispConstants (LIST NIL T))
       (cyclicControlStructures '(WHILE1 WHILEALL WHILENEXT FOR1 FORALL))
       (reservedRuleWords '(self ruleApplied))
       (sendSpellings '(_!
                        _ SEND Send send))
       (stopSpellings '(STOP Stop stop))
       (thenSpellings '(THEN Then
                             then ->))
       (ifSpellings '(IF If if]
)

;; Globals for the RuleSet compiler.

(SETQ GLOBALVARS (APPEND GLOBALVARS RULEVARS))

(PUTPROPS LOOPSRULESP COPYRIGHT ("Xerox Corporation" 1985 1987 1988))
```

FUNCTION INDEX

CONSTANT INDEX

VARIABLE INDEX

PROPERTY INDEX