File created: 11-Jul-88 21:50:16 **{POGO:AISNORTH:XEROX}<LOOPSCORE>LYRIC>USERS>RULES>LOOPSRULESC.;4**

changes to: (FNS CompileRuleList)

previous date: 20-May-88 17:19:54 {POGO:AISNORTH:XEROX}<LOOPSCORE>LYRIC>USERS>RULES>LOOPSRULESC.;3

Read Table: INTERLISP

Package: INTERLISP

Format: XCCS

```
;;
;; Copyright (c) 1985, 1987, 1988 by Xerox Corporation.  All rights reserved.

(RPAQQ LOOPSRULESCCOMS ((DECLARE%: DONTCOPY (PROP MAKEFILE-ENVIRONMENT LOOPSRULESC))
                                                                ; Copyright (c) 1982 by Xerox Corporation
                                                                ; Written in August 1982 by Mark Stefik, Alan Bell, and Danny
                                                                ; Bobrow
                                                                ; Fns for compiling LOOPS RuleSets.
                        (FNS * RULECOMPILEFNS)
                        (VARS (ASSUMELISPFLG))))

(DECLARE%: DONTCOPY

(PUTPROPS LOOPSRULESC 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 compiling LOOPS RuleSets.

(RPAQQ RULECOMPILEFNS
       (AuditRecordCodeGen BreakLHSCodeGen BreakRHSCodeGen CheckVariableNameConflict CompileAssnStmnt
              CompileComment CompileCompositeGetTerm CompileCompositePutTerm CompileExpr CompileGetTerm
              CompileLHS CompileOpPrecedenceExpr CompileParenExpr CompilePopStmnt CompilePopTerms
              CompilePropGetTerm CompilePropPutTerm CompilePushStmnt CompilePushTerm CompilePutTerm
              CompileQuotedConstant CompileRHS CompileRule CompileRule1 CompileRuleList CompileRuleOrLabel
              ContainsItem ExprCodeGen FPrecedence ForTemplate GPrecedence GetRuleStrings OSSetCode OSTestCode
              TraceLHSCodeGen TraceRHSCodeGen))

(DEFINEQ

(AuditRecordCodeGen
  [LAMBDA NIL                                                   (* RBGMartin "18-Feb-87 15:06")

          (* * Subroutine of CompileRHS. Generates LISP code for creating an audit record for the current rule.)

    (PROG [auditVars setStatements allSpecifications code metaVal
           (auditTemplate (CONSTANT '(PROGN                     (* Make an audit record for this rule and set its audit values.)
                                          (SETQ ^auditRecord (_ ($ ^auditClassName)
                                                                New))
                                          ^setStatements]
          (SETQ allSpecifications (APPEND ruleAuditSpecification auditSpecification))
          (for assnVar in allSpecifications unless (FMEMB (CAR assnVar)
                                                          auditVars)
             do (SETQ auditVars (CONS (CAR assnVar)
                                      auditVars)))
          [SETQ setStatements (for assnVar in auditVars collect [SETQ metaVal (CompileGetTerm (CDR (ASSOC assnVar
                                                                                                      allSpecifications
                                                                                                      ]
                                                                                (LIST 'PutValue '^auditRecord (KWOTE assnVar)
                                                                                      (COND
                                                                                         [(Object? metaVal)
                                                                                          (LIST 'GetObjFromUID (KWOTE (UID
                                                                                                                        ruleObject
                                                                                                                        ]
                                                                                         (T metaVal]
          (SETQ code (SUBST (ClassName rsAuditClass)
                            '^auditClassName auditTemplate))
          (SETQ code (LSUBST setStatements '^setStatements code))
          (RETURN code])

(BreakLHSCodeGen
  [LAMBDA NIL                                                   (* RBGMartin "18-Feb-87 14:10")

          (* * Generates varCode for tracing variables and breaking when the LHS of a rule is tested.)

    (PROG (varCode traceCode ruleCode code)

          (* * Compute the parts of the break code.)

          [COND
```

```
      (debugVars (SETQ varCode (CONS 'WRITETTY (for var in debugVars
                                                  join (LIST " " (CONCAT (UnParseTerm var)
                                                                         "=")
                                                             (CompileGetTerm var]
   (SETQ ruleCode (LIST '_ (LIST 'GetObjFromUID (KWOTE (UID ruleObject)))
                        'Print))
   (SETQ traceCode (LIST 'TraceLHS (KWOTE rsName)
                         (KWOTE ruleLabel)
                         ruleNumber))
```

(* * Splice the code together.)

```
   [SETQ code (LIST (LIST 'RE]
   [COND
      (varCode (SETQ code (CONS varCode code]
   [COND
      (ruleCode (SETQ code (CONS ruleCode code]
   [SETQ code (CONS 'PROGN (CONS '                          (* Rule Breaking and Tracing Code)
                                 (CONS traceCode code]
   (RETURN code])
```

(BreakRHSCodeGen

```
  [LAMBDA NIL                                               (* RBGMartin "18-Feb-87 14:11")
```

(* * Generates varCode for tracing variables and breaking when the RHS of a rule is executed.)

```
    (PROG (varCode traceCode ruleCode code)
```

(* * Compute the parts of the break code.)

```
   [COND
      (debugVars (SETQ varCode (CONS 'WRITETTY (for var in debugVars
                                                  join (LIST " " (CONCAT (UnParseTerm var)
                                                                         "=")
                                                             (CompileGetTerm var]
   [COND
      ((EQ rsBreakFlg 'B)
```

(* If B options, generate code to print rule. Skip if BT since LHS trace already prints the rule.)

```
       (SETQ ruleCode (LIST '_ (LIST 'GetObjFromUID (KWOTE (UID ruleObject)))
                            'Print]
   (SETQ traceCode (LIST 'TraceRHS (KWOTE rsName)
                         (KWOTE ruleLabel)
                         ruleNumber))
```

(* * Splice the code together.)

```
   [SETQ code (LIST (LIST 'RE]
   [COND
      (varCode (SETQ code (CONS varCode code]
   [COND
      (ruleCode (SETQ code (CONS ruleCode code]
   [SETQ code (CONS 'PROGN (CONS '                          (* Rule Breaking and Tracing Code)
                                 (CONS traceCode code]
   (RETURN code])
```

(CheckVariableNameConflict

```
  [LAMBDA NIL                                               (* smL "19-Dec-85 16:40")
                                                            (* Checks that the names of variables in different categories are
```

unique.)

```
    (PROG [badVars (varLists (CONSTANT '(wsClassVars wsVars taskVars tempVars lispVars]
          (for vl1 in varLists when (BOUNDP vl1) do (for vl2 in varLists unless (OR (NOT (BOUNDP vl2))
                                                                                    badVars
                                                                                    (EQ vl1 vl2))
                                                                                do (COND
                                                                                      ((SETQ badVars (INTERSECTION (EVALV vl1)
                                                                                                                   (EVALV vl2)))
                                                                                       (SETQ parseErrorFlg T)
                                                                                       (FlushRule "Variable name conflict for " badVars])
```

(CompileAssnStmnt

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

(* * Parse an assignment statement in a rule. Subroutine of Expr.
Input is in the global variable ruleSetTokens. The subroutine is expected to remove the ruleSetTokens that it recognizes, and to return LISP code as its value.)

```
    (PROG (assnVar assnVal)
```

(* * Get the assignment variable.)

```
    (SETQ assnVar (pop ruleSetTokens))
```

```
          (* * Pop the _)

          (pop ruleSetTokens)

          (* * Parse the assignment value.)

          (SETQ assnVal (CompileExpr))

          (* * Parse the assignment variable.)

          (RETURN (CompilePutTerm assnVar assnVal])
```

(CompileComment

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

          (* * Parse a comment. Subroutine of ParseRuleList. Input is in the global variable ruleSetTokens.
          The subroutine is expected to remove the ruleSetTokens that it recognizes, and to return LISP code as its value.)
                                                                 (* Collect all ruleSetTokens until reaching the closing right
                                                                 parenthesis.)
    (PROG (token doneFlg commentCode)
          (pop ruleSetTokens)
          [repeatuntil doneFlg do (SETQ token (pop ruleSetTokens))
                                  (COND
                                     ((EQ token lpar)            (* Recur for embedded parens.)
                                      (push commentCode (CompileComment)))
                                     ((EQ token rpar)            (* quit after right paren.)
                                      (SETQ doneFlg T)
                                      (SETQ token asterisk))
                                     (T                          (* otherwise collect the token.)
                                        (push commentCode token]
          (SETQ commentCode (DREVERSE commentCode))
          (SELECTQ controlType
               ((DO1 CYCLE1)                                     (* For cycle1 and do1, embed the comments in NO-OP clauses
                                 for the COND statement.)
                    (SETQ commentCode (LIST NIL commentCode NIL)))
               NIL)
          (RETURN commentCode])
```

(CompileCompositeGetTerm

```
  [LAMBDA (term)                                                 (* mjs%: " 8-JUN-83 10:51")

          (* * Compile Composite Get forms of an expression in a rule. Sub of CompileGetTerm.)

    (SELECTQ (CAR term)
         (%.                                                     (* A.B = (_ A B))
             (LIST leftArrow (CompileGetTerm (CADR term))
                   (CADDR term)))
         (\                                                      (* Lisp variables.)
            (CADDR term))
         (.! (LIST 'DoMethod (CompileGetTerm (CADR term))
                   (CompileGetTerm (CADDR term))))
         ((%: %:!)                                               (* a%:b and a%:!b)
             [LIST 'GetValue (CompileGetTerm (CADR term))
                   (COND
                      ((EQ (CAR term)
                           colon)
                       (KWOTE (CADDR term)))
                      ((EQ (CAR term)
                           colonBang)
                       (CompileGetTerm (CADDR term])
         ((|::| |::!|)                                           (* |a::b| and |a::!b|)
             [LIST 'GetClassValue (CompileGetTerm (CADR term))
                   (COND
                      ((EQ (CAR term)
                           coloncolon)
                       (KWOTE (CADDR term)))
                      ((EQ (CAR term)
                           colonColonBang)
                       (CompileGetTerm (CADDR term])
         (|..|                                                   (* A..B = (RunRS A B))
              (LIST 'RunRS [KWOTE (CADR (CompileGetTerm (CADR term]
                    (CompileGetTerm (CADDR term))))
         (.,                                                     (* A.,B = (_ ($ Task) New A B ^rs))
             (LIST leftArrow '($ Task)
                   'New
                   (CompileGetTerm (CADR term))
                   (CompileGetTerm (CADDR term))
                   '^rs))
         (..*                                                    (* A..*B is transfer call for A..B)
              (LIST 'RuleSetTransfer (LIST 'LIST (CompileGetTerm (CADR term))
                                           (CompileGetTerm (CADDR term))
                                           '^rs)))
         ((%:, %:,!)                                             (* property values.)
             (CompilePropGetTerm term))
         ($                                                      (* Loops term.)
```

```
          term)
       (FlushRule "Bad Term " term])
```

(CompileCompositePutTerm

```
[LAMBDA (term assnVal)                                    (* mjs%: " 8-JUN-83 11:33")
```

(* * Subroutine of CompilePutTerm. Handles cases of composite term.)

```
   (COND
      ((EQ (CAR term)
           backSlash)                                     (* lisp vars.)
       (LIST 'SETQ (CADDR term)
             assnVal))
      [(FMEMB (CAR term)
              (CONSTANT (LIST colon colonBang)))          (* a%:b and a%:!b)
       (COND
          ((AND ruleAuditFlg ruleRHSFlg)
           (SETQ ruleNeedsAuditFlg T)
           (LIST 'PutAuditRec (CompileGetTerm (CADR term))
                 [COND
                    ((EQ (CAR term)
                         colon)
                     (KWOTE (CADDR term)))
                    ((EQ (CAR term)
                         colonBang)
                     (CompileGetTerm (CADDR term]
                 assnVal
                 '^auditRecord))
          (T (LIST 'PutValue (CompileGetTerm (CADR term))
                   [COND
                      ((EQ (CAR term)
                           colon)
                       (KWOTE (CADDR term)))
                      ((EQ (CAR term)
                           colonBang)
                       (CompileGetTerm (CADDR term]
                   assnVal]
      [(FMEMB (CAR term)
              (CONSTANT (LIST coloncolon colonColonBang)))  (* |a::b| and |a::!b|)
       (COND
          ((AND ruleAuditFlg ruleRHSFlg)
           (SETQ ruleNeedsAuditFlg T)
           (LIST 'PutClassAuditRec (CompileGetTerm (CADR term))
                 [COND
                    ((EQ (CAR term)
                         colon)
                     (KWOTE (CADDR term)))
                    ((EQ (CAR term)
                         colonBang)
                     (CompileGetTerm (CADDR term]
                 assnVal
                 '^auditRecord))
          (T (LIST 'PutClassValue (CompileGetTerm (CADR term))
                   [COND
                      ((EQ (CAR term)
                           coloncolon)
                       (KWOTE (CADDR term)))
                      ((EQ (CAR term)
                           colonColonBang)
                       (CompileGetTerm (CADDR term]
                   assnVal]
      ((FMEMB (CAR term)
              (CONSTANT (LIST colonComma colonCommaBang)))  (* property values.)
       (CompilePropPutTerm term assnVal))
      (T (FlushRule "Bad Term on left in Assignment Statement:" term)
         NIL])
```

(CompileExpr

```
[LAMBDA NIL                                               (* mjs%: "27-JAN-83 00:50")
```

(* * Parse an expression in a rule. Subroutine of ParseLHS and ParseRHS.
Input is in the global variable ruleSetTokens. The subroutine is expected to remove the ruleSetTokens that it recognizes, and to return LISP code as its value.)

```
   (COND
      (parseErrorFlg (SkipRule))
      ((AND (EQ (CADR ruleSetTokens)
                leftArrow)
            (NEQ (CAR ruleSetTokens)
                 lpar))                                   (* Here for Assignment Statements.)
       (CompileAssnStmnt))
      ((EQ (CADR ruleSetTokens)
           push)                                          (* Here for Push Statements.)
       (CompilePushStmnt))
      ((EQ (CADR ruleSetTokens)
```

```
          pop)                                              (* Here for Pop Statements.)
         (CompilePopStmnt))
        ((EQ (CAR ruleSetTokens)
             quoteSign)                                     (* Here for Quoted Constants.)
         (CompileQuotedConstant))
        ((OR (EQ (CAR ruleSetTokens)
                 semicolon)
             (FMEMB (CAR ruleSetTokens)
                    thenSpellings))                         (* Error.)
         (FlushRule "Unexpected " (CAR ruleSetTokens)))
        (T                                                  (* Here for operator precedence part of expression grammar.)
           (CompileOpPrecedenceExpr])
```

(CompileGetTerm

```
  [LAMBDA (term)                                            (* dgb%: "21-Feb-84 15:21")
```

(* * Compile the Get form of a term in an expression in a rule. Subroutine of ParseTerm, and recursive subroutine of CompileGetTerm and ParsePutTerm. Input is in argument term.
The subroutine is expected to return LISP code as its value. Generates code to Get the value of a term.)

```
    (COND
       ((NUMBERP term)                                      (* number = number)
        term)
       ((STRINGP term)                                      (* Just return strings)
        term)
       ((LITATOM term)
        (COND
           ((FMEMB term taskVars)                           (* taskVars = (GetValue ^task (QUOTE term)))
            (LIST 'GetValue '^task (KWOTE term)))
           ((FMEMB term compileTimeVars)                    (* Compile-time vars get evaluated now.)
            (EVALV term))
           ((NOT (LetterP term))
            (FlushRule "Unrecognized term:" term))
           ((FMEMB term wsVars)
            (LIST 'GetValue 'self (KWOTE term)))
           ((OR (FMEMB term tempVars)
                (FMEMB term lispConstants)
                (FMEMB term reservedRuleWords)
                (FMEMB term rsArgs)
                (GETPROP term 'CLISPWORD))                  (* tempVars and lispVars and lispConstants and reserved words
            = term)
            term)
           (T (OR ASSUMELISPFLG (printout PPDefault T T "*** Assuming that " .FONT BOLDFONT term .FONT
                                          DEFAULTFONT " is a Lisp variable." T))
              term)))
       ((NLISTP term)                                       (* Error if not list.)
        (FlushRule "Unrecognized Term:" term))
       (T                                                   (* Here for composite terms.)
          (CompileCompositeGetTerm term])
```

(CompileLHS

```
  [LAMBDA (augmentedFlg newFlg specialFlg)                  (* mjs%: "12-FEB-83 12:45")
```

(* * Parse the LHS of a rule. Subroutine of ParseRule. Input is in the global variable ruleSetTokens.
The subroutine is expected to remove the ruleSetTokens that it recognizes, and to return LISP code as its value.
The variable augmentedFlg is T if this rule has only implicit conditions added by the compiler.
The variable newFlg means to always recompile the lhs -- even if code already exists for it.
specialFlg is set if this is being called to compile an expression that is not really a LHS of a rule --
e.g., a while condition.)

```
    (PROG (exprCode)
          (SETQ ruleRHSFlg NIL)
```

(* * Collect the code from each of the exprs of the LHS.)

```
          [COND
             ((NULL augmentedFlg)                           (* Parse up to THEN, unless this is an augmented LHS)
              (SETQ exprCode (collect (CompileExpr) until (OR parseErrorFlg (COND
                                                                   ((EQ (CAR ruleSetTokens)
                                                                        semicolon)
                                                                    (FlushRule "Missing" "THEN" "in
                                                                           rule")
                                                                    (SETQ parseErrorFlg T)))
                                                             (FMEMB (CAR ruleSetTokens)
                                                                    thenSpellings]
          [COND
             ((NOT specialFlg)
              [COND
                 [oneShotBangFlg                            (* Add two extra LHS clauses if this is a one-shot-bang rule.)
                        (SETQ exprCode (CONS (OSTestCode)
                                             (CONS (OSSetCode)
                                                   exprCode]
                 (oneShotFlg                                (* Add extra LHS clause if this is a one-shot rule.)
                        (SETQ exprCode (CONS (OSTestCode)
                                             exprCode]
```

```
            [COND
               ((EQ ruleTraceFlg 'TT)                          (* Add extra LHS clause if Rule Tracing is requested on test.)
                (SETQ exprCode (CONS (TraceLHSCodeGen)
                                     exprCode]
            (COND
               ((EQ ruleBreakFlg 'BT)                          (* Add extra LHS clause if Rule Test Breaking is requested.)
                (SETQ exprCode (CONS (BreakLHSCodeGen)
                                     exprCode]

      (* * Return the LHS code.)

      [SETQ exprCode (COND
                        (parseErrorFlg                         (* NIL on parsing error.)
                               NIL)
                        ((EQP (FLENGTH exprCode)
                              1)                               (* One expr = <expr>)
                         (CAR exprCode))
                        (T                                     (* Several expressions = (AND <expr> <expr> |...|))
                           (CONS 'AND exprCode]
      (RETURN exprCode])
```

(CompileOpPrecedenceExpr

```
  [LAMBDA NIL                                                  (* mjs%: "17-MAR-83 14:15")
```

(* * Parse the operator precedence part expression in a rule. Subroutine of ParseExpr.
Input is in the global variable ruleSetTokens. The subroutine is expected to remove the ruleSetTokens that it recognizes,
and to return LISP code as its value.)

```
    (PROG [nextToken prevType term oprStack argStack (endTokens (CONSTANT (LIST semicolon rightArrow)))
           (specOprs (CONSTANT (LIST unaryMinus notSign lbracket)))
           (operators (CONSTANT (LIST lbracket rbracket plus minus plusplus minusminus unaryMinus asterisk
                                      slash lessSign greaterSign lessEqSign greaterEqSign eqSign eqeqSign
                                      notSign neqSign membSign leftArrow endExpr]
                                                               (* Initialize stacks for operators and operands.)
      (SETQ oprStack (LIST endExpr))
    NextToken
      (SETQ nextToken (CAR ruleSetTokens))
      (COND
         (parseErrorFlg (GO QUIT)))
      [COND
         ((OR (FMEMB nextToken endTokens)
              (FMEMB nextToken thenSpellings))
          (GO LastReduceLoop))
         ((NOT (FMEMB nextToken operators))                    (* Here for operand.)
          (COND
             ((EQ prevType 'operand)                           (* Two sequential operands -> end of expr.)
              (GO LastReduceLoop))
             ((EQ (CAR ruleSetTokens)
                  quoteSign)                                   (* Here for quoted constant.)
              (SETQ term (CompileQuotedConstant))
              (push argStack term)
              (SETQQ prevType operand)
              (GO NextToken))
             ((EQ (CAR ruleSetTokens)
                  lpar)                                        (* Here for Fn Call etc.)
              (SETQ term (CompileParenExpr))
              (push argStack term)
              (SETQQ prevType operand)
              (GO NextToken))
             (T                                                (* Here for term.)
                (SETQ term (CompileGetTerm (pop ruleSetTokens)))
                (push argStack term)
                (SETQQ prevType operand)
                (GO NextToken]                                 (* Here for operator)
      (COND
         ((AND (FMEMB nextToken specOprs)
               (EQ prevType 'operand))                         (* operand followed by unary opr or bracket expression -> end
                                                               of expr.)
          (GO LastReduceLoop)))
      (SETQ prevType 'operator)
      (COND
         ((ILESSP (FPrecedence (CAR oprStack))
                  (GPrecedence nextToken))                     (* Shift!)
          (push oprStack (pop ruleSetTokens)))
         (T                                                    (* Reduce!)
            (ExprCodeGen)))
      (GO NextToken)

      (* * Here if no more ruleSetTokens in the expression.)

    LastReduceLoop
      (COND
         (parseErrorFlg (GO QUIT)))
      (COND
         ((AND (NULL (CADR oprStack))
               (NULL (CADR argStack)))                         (* Normal Finish)
```

```
            (GO QUIT)))
          (ExprCodeGen)
          (GO LastReduceLoop)
      QUIT
          (RETURN (CAR argStack])
```

(CompileParenExpr

```
  [LAMBDA NIL                                                    (* dgb%: "21-Feb-84 15:21")
```

(* * Parse a parenthesized clause -- either a LISP function call or a LOOPS SEND message statement or a STOP statement -- in a rule. Subroutine of Expr. Input is in the global variable ruleSetTokens.
The subroutine is expected to remove the ruleSetTokens that it recognizes, and to return LISP code as its value.)

```
    (PROG (fnName object selector args)
```

(* * Pop the left parenthesis and get the function name.)

```
          (pop ruleSetTokens)
          (SETQ fnName (pop ruleSetTokens))
```

(* * Collect the function name or SEND preface.)

```
          [COND
             ((FMEMB fnName sendSpellings)                       (* msg.)
              (SETQ object (CompileExpr))
              (SETQ selector (COND
                                 ((EQ fnName leftArrowBang)      (* For _! msg evaluate the selector.)
                                  (CompileExpr))
                                 (T                              (* For _ msg do not eval selector.)
                                    (pop ruleSetTokens]
```

(* * Collect the Arguments.)

```
          (SETQ args (while (AND ruleSetTokens (NEQ (CAR ruleSetTokens)
                                                    rpar)
                                 (NEQ (CAR ruleSetTokens)
                                      semicolon))
                        collect (CompileExpr)))
```

(* * pop the right parenthesis.)

```
          (COND
             ((NEQ (CAR ruleSetTokens)
                   rpar)
              (FlushRule "Missing right parenthesis"))
             (T (pop ruleSetTokens)))
```

(* * Return the Function, STOP statement, or Msg expression.)

```
          (RETURN (COND
                     [(EQ fnName leftArrowBang)                  (* _! msg.)
                      (CONS 'DoMethod (CONS object (CONS selector (CONS NIL args]
                     [(FMEMB fnName sendSpellings)               (* _ msg)
                      (CONS leftArrow (CONS object (CONS selector args]
                     [(FMEMB fnName stopSpellings)               (* STOP statement)
                      (LIST 'PROGN '                             (* ^value set by RuleSetStop)
                            (CONS 'RuleSetStop args)
                            (LIST 'GO 'QUIT]
                     (T                                          (* LISP fn)
                        (COND
                           ([AND (NOT (FNTYP fnName))
                                 (NOT (GETPROP fnName 'CLISPWORD]
                            (printout PPDefault T T "*** Warning Unrecognized LISP fn " .FONT BOLDFONT fnName
                                  .FONT DEFAULTFONT T)))
                        (CONS fnName args])
```

(CompilePopStmnt

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

(* * Parse a pop statement in a rule. Subroutine of Expr. Input is in the global variable ruleSetTokens.
The subroutine is expected to remove the ruleSetTokens that it recognizes, and to return LISP code as its value.)

```
    (PROG (assnVar stackVar)                                     (* Get the assignment variable.)
          (SETQ assnVar (pop ruleSetTokens))                     (* Pop the _- token.)
          (pop ruleSetTokens)                                    (* Get the pop value.)
          (SETQ stackVar (pop ruleSetTokens))                    (* Compile it.)
          (RETURN (CompilePopTerms assnVar stackVar])
```

(CompilePopTerms

```
  [LAMBDA (assnVar stackVar)                                     (* mjs%: "26-OCT-82 11:17")
```

(* * Parse the terms in a pop statement. The subroutine is expected to return LISP code as its value.
Generates code to Pop the first item of the list stored as the value of stackVar, update the stack, and return the item.)

```
    (PROG [getStackCode putStackCode putItemCode (popTemplate (CONSTANT '(PROG (^item ^stack)
                                                                    (* Pop Statement. ^assnVar _- ^stackVar)
                                                                        (SETQ ^stack ^getStackCode)
                                                                        (SETQ ^item (CAR ^stack))
                                                                        (SETQ ^stack (CDR ^stack))
                                                                    ^putStackCode
                                                                    ^putItemCode
                                                                        (RETURN ^item]

          (* * Generate code for getting the stack, storing the revised stack, and storing the item.)

          (SETQ getStackCode (CompileGetTerm stackVar))
          (SETQ putStackCode (CompilePutTerm stackVar '^stack))
          (SETQ putItemCode (CompilePutTerm assnVar '^item))

          (* * Substitute the code into the pop template.)

          (SETQ popTemplate (SUBST getStackCode '^getStackCode popTemplate))
          (DSUBST putStackCode '^putStackCode popTemplate)
          (DSUBST putItemCode '^putItemCode popTemplate)
          (DSUBST assnVar '^assnVar popTemplate)
          (DSUBST stackVar '^stackVar popTemplate)

          (* * Return the instantiated template for the pop statement.)

          (RETURN popTemplate])
```

(CompilePropGetTerm

```
  [LAMBDA (term)                                                    (* mjs%: " 8-JUN-83 10:49")

          (* * Subroutine of CompileCompositeGetTerm. Handles the property values cases for a%:,b |a:b:,c| etc.)

    (COND
       [(LISTP (CADR term))
        (PROG (term1)
              (SETQ term1 (CADR term))
              (RETURN (COND
                         ((FMEMB (CAR term1)
                                 (CONSTANT (LIST colon colonBang coloncolon colonColonBang)))
                                                                    (* colon/coloncolon combinations)
                          (LIST (COND
                                   ((FMEMB (CAR term1)
                                           (CONSTANT (LIST coloncolon colonColonBang)))
                                    'GetClassValue)
                                   (T 'GetValue))
                                (CompileGetTerm (CADR term1))
                                [COND
                                   ((FMEMB (CAR term1)
                                           (CONSTANT (LIST colon coloncolon)))
                                    (KWOTE (CADDR term1)))
                                   (T (CompileGetTerm (CADDR term1]
                                (COND
                                   ((EQ colonComma (CAR term))
                                    (KWOTE (CADDR term)))
                                   ((EQ colonCommaBang (CAR term))
                                    (CompileGetTerm (CADDR term]
       ((OR (FMEMB (CADR term)
                   ruleVars)
            (FMEMB (CADR term)
                   compileTimeVars))                                (* Bad property request.)
        (FlushRule "Illegal to fetch property of this var: " term))
       (T                                                           (* A%:,B = (GetValue self (QUOTE A)
                                                                    (QUOTE B)))
          (LIST 'GetValue 'self (KWOTE (CADR term))
                (COND
                   ((EQ (CAR term)
                        colonComma)                                 (* a%:,b)
                    (KWOTE (CADDR term)))
                   ((EQ (CAR term)
                        colonCommaBang)                             (* a%:,!b)
                    (CompileGetTerm (CADDR term])
```

(CompilePropPutTerm

```
  [LAMBDA (term assnVal)                                            (* mjs%: " 8-JUN-83 10:49")

          (* * Subroutine of CompilePutTerm. Handles the cases for assignment to property values as in a%:,b etc.)

    (COND
       [(LISTP (CADR term))
        (PROG (term1)
              (SETQ term1 (CADR term))
              (RETURN (COND
                         [(FMEMB (CAR term1)
                                 (CONSTANT (LIST colon colonBang coloncolon colonColonBang)))
                                                                    (* |a:b:,c| and |a:!b:,c| and |a:b:,!c| and |a:!b:,!c|)
```

```
                        (LIST (COND
                                 ((FMEMB (CAR term1)
                                         (CONSTANT (LIST coloncolon colonColonBang)))
                                  'PutClassValue)
                                 (T 'PutValue))
                              (CompileGetTerm (CADR term1))
                              [COND
                                 ((FMEMB (CAR term1)
                                         (CONSTANT (LIST colon coloncolon)))
                                  (KWOTE (CADDR term1)))
                                 ((EQ (CAR term1)
                                      colonBang)
                                  (CompileGetTerm (CADDR term1]
                              assnVal
                              (COND
                                 ((EQ (CAR term)
                                      colonComma)
                                  (KWOTE (CADDR term)))
                                 ((EQ (CAR term)
                                      colonCommaBang)
                                  (CompileGetTerm (CADDR term]
                        (T (FlushRule "Invalid property access term" term]
    ((OR (FMEMB (CADR term)
                ruleVars))                                    (* Errors.)
     (FlushRule "Illegal to store property of RuleVar:" term))
    (T                                                        (* a%:,b and a%:,!b)
       (LIST 'PutValue 'self (KWOTE (CADR term))
             assnVal
             (COND
                ((EQ (CAR term)
                     colonComma)
                 (KWOTE (CADDR term)))
                ((EQ (CAR term)
                     colonCommaBang)
                 (CompileGetTerm (CADDR term])
```

(CompilePushStmnt

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

(* * Parse a push statement in a rule. Subroutine of Expr. Input is in the global variable ruleSetTokens.
The subroutine is expected to remove the ruleSetTokens that it recognizes, and to return LISP code as its value.)

```
  (PROG (assnVal stackVar)                                    (* Get the assignment variable.)
        (SETQ stackVar (pop ruleSetTokens))                   (* Pop the _+)
        (pop ruleSetTokens)                                   (* Parse the push value.)
        (SETQ assnVal (CompileExpr))                          (* Parse the push variable.)
        (RETURN (CompilePushTerm stackVar assnVal])
```

(CompilePushTerm

```
[LAMBDA (stackVar assnVal)                                    (* mjs%: "26-OCT-82 11:44")
```

(* * Parse the terms in a push statement. The subroutine is expected to return LISP code as its value.
Generates code to Push the assnVal onto the list stored as the value of stackVar and return the item.)

```
  (PROG [getStackCode putStackCode putItemCode (pushTemplate (CONSTANT '(PROG (^stack ^item)
                                                              (* Push Statement.)
                                                                    (SETQ ^item ^assnVal)
                                                                    (SETQ ^stack (CONS ^item
                                                                                       ^getStackCode
                                                                                       ))
                                                                ^putStackCode
                                                                    (RETURN ^item]
```

(* * Generate code for getting the stack, storing the revised stack, and storing the item.)

```
        (SETQ getStackCode (CompileGetTerm stackVar))
        (SETQ putStackCode (CompilePutTerm stackVar '^stack))
```

(* * Substitute the code into the push template.)

```
        (SETQ pushTemplate (SUBST getStackCode '^getStackCode pushTemplate))
        (DSUBST putStackCode '^putStackCode pushTemplate)
        (DSUBST assnVal '^assnVal pushTemplate)
```

(* * Return the instantiated template for the pop statement.)

```
        (RETURN pushTemplate])
```

(CompilePutTerm

```
[LAMBDA (term assnVal)                                        (* mjs%: " 9-JUN-83 13:30")
```

(* * Compile the Put form of a term in an expression in a rule. Input is in the argument term.
The subroutine is expected to return LISP code as its value. Generates code to Put the value of assnVal into term.)

```
    (COND
       ((NUMBERP term)                                          (* number = error)
        (FlushRule "Unexpected Number on left in Assignment Statement:" term))
       [(LITATOM term)
        (COND
           ((OR (FMEMB term tempVars)
                (FMEMB term rsArgs))                            (* tempVars)
            (LIST 'SETQ term assnVal))
           ((FMEMB term taskVars)                               (* taskVars)
            (LIST 'PutValue '^task (KWOTE term)
                  assnVal))
           ((OR (FMEMB term compileTimeVars)
                (FMEMB term reservedRuleWords))
            (FlushRule "Unexpected constant on left in Assignment Statement:" term))
           ((NOT (LetterP term))
            (FlushRule "Unexpected " term))
           ((NOT (FMEMB term wsVars))
            (FlushRule "Unrecognized IV:" term))
           ((AND ruleAuditFlg ruleRHSFlg)                       (* work space variable audited)
            (SETQ ruleNeedsAuditFlg T)
            (LIST 'PutAuditRec 'self (KWOTE term)
                  assnVal
                  '^auditRecord))
           (T                                                   (* work space variable.)
              (LIST 'PutValue 'self (KWOTE term)
                    assnVal]
       ((NLISTP term)                                           (* Error if not list.)
        (FlushRule "Bad Term on left in Assignment Statement" term))
       (T                                                       (* Composite put term.)
          (CompileCompositePutTerm term assnVal])
```

(CompileQuotedConstant

```
  [LAMBDA NIL                                                   (* mjs%: "22-JAN-83 10:46")
```

(* * Parse a quoted constant in a rule. Subroutine of Expr. Input is in the global variable ruleSetTokens.
The subroutine is expected to remove the ruleSetTokens that it recognizes, and to return LISP code as its value.)

```
    (PROG [token qTokens doneFlg (qEndTokens (CONSTANT (LIST rpar semicolon]
                                                                (* Discard the quoteSign)
          (pop ruleSetTokens)                                   (* Get the first token of the quoted constant.)
          (SETQ token (pop ruleSetTokens))
          (COND
             ((NEQ token lpar)                                  (* |...| %' NL |...| -> (QUOTE NL))
              (RETURN (KWOTE token)))
             (T                                                 (* |...| %' (A1 A2 A3 |...|) |...| -> (QUOTE
                                                                (A1 A2 A3 |...|)))
                (SETQ qTokens (eachtime (SETQ token (pop ruleSetTokens)) until (FMEMB token qEndTokens)
                                collect token))
                (COND
                   ((EQ token semicolon)
                    (FlushRule "Unexpected" "semicolon" "in quoted constant")))
                (RETURN (KWOTE qTokens])
```

(CompileRHS

```
  [LAMBDA NIL                                                   (* mjs%: " 7-JUN-83 11:16")
```

(* * Parse the RHS of a rule. Subroutine of ParseRule. Input is in the global variable ruleSetTokens.
The subroutine is expected to remove the ruleSetTokens that it recognizes, and to return LISP code as its value.)

```
    (PROG (token exprCode)
          (SETQ ruleRHSFlg T)
```

(* * Here to compile RHS)

```
          (COND
             ((FMEMB (CAR ruleSetTokens)
                     thenSpellings)                             (* Pop the THEN.)
              (pop ruleSetTokens)))
```

(* * Collect the code from each of the exprs of the RHS.)

```
          [SETQ exprCode (collect (CompileExpr) until (OR parseErrorFlg (EQ (CAR ruleSetTokens)
                                                                            semicolon]
          [COND
             ((AND oneShotFlg (NULL oneShotBangFlg))
```

(* Add extra RHS clause if this is a one-shot rule but not a one-shot-bang rule.)

```
              (SETQ exprCode (CONS (OSSetCode)
                                   exprCode]
          [COND
             (ruleTraceFlg                                      (* Here to trace a rule.)
                    (SETQ exprCode (CONS (TraceRHSCodeGen)
                                         exprCode)))
             (ruleTraceFlg                                      (* Here to trace if rule is tested.)
```

```
                (SETQ exprCode (CONS (LIST 'TraceRHS (KWOTE ruleLabel)
                                           ruleNumber)
                                     exprCode]
        [COND
           (ruleBreakFlg                                          (* Add extra RHS clause if rule tracing is requested.)
                  (SETQ exprCode (CONS (BreakRHSCodeGen)
                                       exprCode]
        [COND
           (ruleNeedsAuditFlg                                     (* Add extra RHS clause if rule needs audit record.)
                  (SETQ exprCode (CONS (AuditRecordCodeGen)
                                       exprCode]

        (* * Return the RHS code.)

        [SETQ exprCode (COND
                          ((EQP (FLENGTH exprCode)
                                1)                                (* One expr = (SETQ ^value <expr>))
                           (CONS 'SETQ (CONS '^value exprCode)))
                          (T                                      (* Several expressions = (SETQ ^value
                                                                  (PROGN <expr> <expr> |...|)))
                             (CONS 'SETQ (CONS '^value (CONS (CONS 'PROGN exprCode]
                                                                  (* Save the compiled code in the rule object.)
        (RETURN exprCode])
```

(CompileRule

```
  [LAMBDA (self ruleSetSource)                                    (* RBGMartin "16-Feb-87 16:55")

        (* * Parse a rule. Subroutine of CompileRuleList. Input is in the global variable ruleSetTokens.
        The subroutine is expected to remove the ruleSetTokens that it recognizes, and to return LISP code as its value.
        Most of the work is done in CompileRule1. Variable self is the current RuleSet.)

    (PROG (ruleCode ruleOldSource)

        (* * Increment rule number.)

        (SETQ ruleNumber (ADD1 ruleNumber))
        (SPACES 1 PPDefault)
        (PRIN1 ruleNumber PPDefault)

        (* * Fetch old rule object, if any.)

        (SETQ ruleObject (CAR (NTH rsRuleObjects ruleNumber)))
        (RETURN (CompileRule1])
```

(CompileRule1

```
  [LAMBDA (specialFlg)                                            (* dgb%: "17-Feb-84 18:08")

        (* * Subroutine of CompileRule and of the RuleExec. Argument specialFlg is T if not compiling a rule in a RuleSet --
        that is, if called by RuleExec.)

    (PROG (firstLastFlg (ruleBndry (CONSTANT (LIST semicolon)))
                ruleForm ruleCode)
        (DECLARE (SPECVARS firstLastFlg))                         (* Interpret any meta information.)
        (GetRuleMetaDecls)
        [SETQ ruleForm (COND
                          ((FMEMB (CAR ruleSetTokens)
                                  ifSpellings)                    (* IF |...| = IfThen)
                           (pop ruleSetTokens)
                           'IfThen)
                          ((FMEMB (CAR ruleSetTokens)
                                  thenSpellings)                  (* -> |...| = OnlyThen)
                           (pop ruleSetTokens)
                           'OnlyThen)
                          ((ScanFor thenSpellings ruleBndry)      (* |...| -> |...| ; = IfThen)
                           'IfThen)
                          (T                                      (* |...| ; = OnlyThen)
                             'OnlyThen]
        [COND
           ((AND (EQ ruleForm 'OnlyThen)
                 (OR ruleBreakFlg ruleTraceFlg oneShotFlg))       (* Augment an OnlyThen rule if OneShot, or Tracing or
                                                                  Breaking.)
            (SETQ ruleForm 'AugmentedOnlyThen]

        (* * Frame the rule code depending on the controlType. If this is a first or last rule, then we always want the simple
        conditional form)

        (SETQ ruleCode (SELECTQ (COND
                                   (firstLastFlg 'DOALL)
                                   (T controlType))
                                ((DO1 WHILE1 FOR1)
                                     (SELECTQ ruleForm
                                         (IfThen (LIST (CompileLHS NIL NIL specialFlg)
                                                       (CompileRHS)))
                                         (OnlyThen (LIST T (CompileRHS)))
                                         (AugmentedOnlyThen
```

```
                                        (LIST (CompileLHS 'Augmented NIL specialFlg)
                                              (CompileRHS)))
                                   (FlushRule "Unrecognized Rule Syntax")))
                              ((DOALL WHILEALL FORALL)
                                   (SELECTQ ruleForm
                                        (IfThen (LIST 'COND (LIST (CompileLHS NIL NIL specialFlg)
                                                                  (CompileRHS))))
                                        (OnlyThen (CompileRHS))
                                        (AugmentedOnlyThen
                                             (LIST 'COND (LIST (CompileLHS 'Augmented NIL specialFlg)
                                                               (CompileRHS))))
                                        (FlushRule "Unrecognized Rule Syntax")))
                              ((DONEXT WHILENEXT)
                                   (SELECTQ ruleForm
                                        (IfThen [LIST ruleNumber (LIST 'COND (LIST (CompileLHS NIL NIL specialFlg)
                                                                                   (CompileRHS]))
                                        (OnlyThen (LIST ruleNumber (CompileRHS)))
                                        (AugmentedOnlyThen
                                             (LIST 'COND (LIST (CompileLHS 'Augmented NIL specialFlg)
                                                               (CompileRHS))))
                                        (FlushRule "Unrecognized Rule Syntax")))
                              (FlushRule "Unrecognized controlType: " controlType)))
          (COND
             ((EQ (CAR ruleSetTokens)
                  semicolon)                                    (* Pop the semicolon.)
              (pop ruleSetTokens)))

          (* * First and last rules are not collected in the ruleSet, and are always done when appicable)

          (SELECTQ firstLastFlg
               (F (SETQ firstRules (NCONC1 firstRules ruleCode)))
               (L (SETQ lastRules (NCONC1 lastRules ruleCode)))
               (RETURN (LIST ruleCode)))
          (RETURN NIL])
```

(CompileRuleList

```
  [LAMBDA (self ruleSetSource)                                  ; Edited 11-Jul-88 20:59 by jrb:

          (* * Subroutine of RuleSet.CompileRules. Argument self is the RuleSet.
          Input ruleSetTokens are in the global variable ruleSetTokens. An instantiated codeTemplate for executing the RuleList is
          returned as value of this subroutine.)

     (PROG (codeTemplate whileTemplate rules progVars firstRules lastRules)
          (DECLARE (SPECVARS firstRules lastRules))             (* Kludge!)
          (SETQ rsInternalTempVars NIL)
          (COND
             ((NULL wsClass)
              (FlushRule "No " " workSpace " "set yet.  ")
              (RETURN NIL)))
          (SETQ ruleNumber 0)

          (* * Parse the Rules and substitute them into the code template.)

          [while (AND (ILEQ ruleNumber rsNumRules)
                      ruleSetTokens
                      (NOT parseErrorFlg))
             do (COND
                   ((OR (EQ (CAR ruleSetTokens)
                            semicolon)
                        (NULL (CAR ruleSetTokens)))
                    (pop ruleSetTokens))
                   (T (SETQ parseErrorFlg NIL)
                      (SETQ rules (NCONC rules (CompileRuleOrLabel self ruleSetSource]
          [COND
             ((AND rsRuleAppliedFlg (EQ controlType 'WHILE1))   (* Add extra rule for WHILE1.)
              (SETQ rules (NCONC1 rules '(T                     (* Here if no rules were executed.)
                                          (SETQ ruleApplied NIL]

          (* * Determine the codeTemplate for the RuleSet.)

          (SETQ codeTemplate (GetRuleSetTemplate))

          (* * Compute the vars for the PROG of the RuleSet.)

          (SETQ progVars (GetProgVars))
          (SETQ codeTemplate (SUBST progVars '^progVars codeTemplate))
          [COND
             ((FMEMB controlType cyclicControlStructures)
              (COND
                 ((NULL rsWhileCondition)
                  (FlushRule "No While Condition specified for " controlType ". Assuming T.")
                  (SETQ rsWhileCondition T)))
              (SETQ ruleSetTokens (NCONC1 rsWhileCondition rightArrow))
              [PROG ((ASSUMELISPFLG T))                         (* Inside Iteration conditions assume Lisp variables when not
                                                                told otherwise)
                    (SETQ whileTemplate (CompileLHS NIL 'New 'Special]
```

```
            (SETQ codeTemplate (SELECTQ controlType
                                       ((FOR1 FORALL)            (* Special case for Fors)
                                            (ForTemplate codeTemplate whileTemplate))
                                       (DSUBST whileTemplate '^whileCondition codeTemplate]
          (SETQ codeTemplate (LSUBST rules '^rules codeTemplate))
          (AND firstRules (SETQ codeTemplate (LSUBST firstRules '^firstRules codeTemplate)))
          (AND lastRules (SETQ codeTemplate (LSUBST lastRules '^lastRules codeTemplate)))
          (_@
           taskVars taskVars)
```

(* * Compile and Substitute the While Condition.)

```
          (CheckVariableNameConflict)
          (RETURN codeTemplate])
```

(CompileRuleOrLabel

```
  [LAMBDA (self ruleSetSource)                                    (* mjs%: " 7-JUN-83 11:20")
```

(* * Parse a rule, a label, or a comment. Subroutine of CompileRuleList.
Input is in the global variable ruleSetTokens. The subroutine is expected to remove the ruleSetTokens that it recognizes, and to return LISP code as its value.)

```
    (SETQ ruleLabel NIL)
    (COND
       ((EQ (CADR ruleSetTokens)
            colon)                                                (* <label>%: |...| Return ^ruleLabel and discard colon.)
        (SETQ ruleLabel (pop ruleSetTokens))
        (pop ruleSetTokens)
        ruleLabel))                                               (* Discard any comments.)
    (FlushComment?)                                               (* Compile the Rule.)
    (CompileRule self ruleSetSource])
```

(ContainsItem

```
  [LAMBDA (item expr)                                             (* dgb%: "21-Feb-84 09:30")
    (COND
       ((NLISTP expr)
        (EQ item expr))
       (T (OR (ContainsItem item (CAR expr))
              (ContainsItem item (CDR expr])
```

(ExprCodeGen

```
  [LAMBDA NIL                                                     (* mjs%: "17-MAR-83 14:08")
```

(* * Code generation function for expression parsing called from ParseOpPrecedenceExpr.
Operates by side effect on implicitly passed variables oprStack, argStack, and nextToken.)

```
    (PROG [operator arg1 arg2 (noPops (CONSTANT (LIST lbracket rbracket)))
                  (unaryOprs (CONSTANT (LIST unaryMinus notSign]
          (SETQ operator (pop oprStack))
          [COND
             ((FMEMB operator unaryOprs)                          (* pop 1 arg for unary oprs.)
              (SETQ arg1 (pop argStack)))
             ((FMEMB operator noPops)                             (* Skip popping for noPops)
              NIL)
             (T                                                   (* pop 2 args for binary oprs.)
                (SETQ arg2 (pop argStack))
                (SETQ arg1 (pop argStack]
```

(* * Reduce the operator.)

```
          (SETQ reducedArg (SELECTQ operator
                                (%[ (COND
                                       ((EQ nextToken rbracket)
                                        (pop ruleSetTokens)
                                        (SETQ nextToken (CAR ruleSetTokens))
                                        (SETQ prevType 'operand)
                                        (RETURN))
                                       (T (FlushRule "Bracket error in expression"))))
                                (%] (FlushRule "Bracket error in expression"))
                                (+ (LIST 'PLUS arg1 arg2))
                                (- (LIST 'DIFFERENCE arg1 arg2))
                                (-1- (LIST 'MINUS arg1))
                                (++ (LIST 'PLUSPLUS arg1 arg2))
                                (-- (LIST 'MINUSMINUS arg1 arg2))
                                (* (LIST 'TIMES arg1 arg2))
                                (/ (LIST 'QUOTIENT arg1 arg2))
                                (< (LIST 'LESSP arg1 arg2))
                                (<= (LIST 'LEQ arg1 arg2))
                                (> (LIST 'GREATERP arg1 arg2))
                                (>= (LIST 'GEQ arg1 arg2))
                                (= (LIST 'EQ arg1 arg2))
                                (== (LIST 'EQUALS arg1 arg2))
                                (~= (LIST 'NEQ arg1 arg2))
                                (<< (LIST 'FMEMB arg1 arg2))
```

```
                                (~ (LIST 'NOT arg1))
                                (_ (CompilePutTerm arg1 arg2))
                                (FlushRule "Unrecognized operator " operator)))
```

(* * Push reduced arg on Opr Stack.)

```
(push argStack reducedArg])
```

(FPrecedence

```
[LAMBDA (operator)                                          (* mjs%: "17-MAR-83 14:04")
```

(* * F-Precedence function for expression parsing. Called by ParseExpr.)

```
  (SELECTQ operator
       (%[ -1)
       (%] 6)
       (+ 3)
       (- 3)
       (-1- 3)
       (++ 3)
       (-- 3)
       (* 5)
       (/ 5)
       (< 1)
       (<= 1)
       (> 1)
       (>= 1)
       (= 1)
       (== 1)
       (~= 1)
       (<< 1)
       (_ 2)
       (~ 5)
       (endExpr -2)
       (FlushRule "Unrecognized operator" operator])
```

(ForTemplate

```
[LAMBDA (codeTemplate whileTemplate)                        (* dgb%: "21-Feb-84 17:47")
  (PROG [(wt (LIST 'SETQ '^value (COND
                                   [(NOT (ContainsItem 'ruleSet whileTemplate))
                                    (APPEND whileTemplate (LIST 'DO 'ruleSet]
                                   (T whileTemplate]
        (RETURN (DSUBST (SUBST (SELECTQ controlType
                                  (FOR1 (LIST 'COND '^rules))
                                  '^rules)
                               'ruleSet wt)
                        '^forLoop codeTemplate])
```

(GPrecedence

```
[LAMBDA (operator)                                          (* mjs%: "17-MAR-83 14:05")
```

(* * G-Precedence function for expression parsing. Called by ParseExpr.)

```
  (SELECTQ operator
       (%[ 6)
       (%] -1)
       (+ 2)
       (- 2)
       (-1- 2)
       (++ 2)
       (-- 2)
       (* 4)
       (/ 4)
       (< 1)
       (<= 1)
       (> 1)
       (>= 1)
       (= 1)
       (== 1)
       (<< 1)
       (~= 1)
       (_ 2)
       (~ 4)
       (endExpr -2)
       (FlushRule "Unrecognized operator" operator])
```

(GetRuleStrings

```
[LAMBDA (sourceRules)                                       (* mjs%: "11-NOV-82 09:59")
```

(* * Partitions the sourceRules into a list of strings into individual statements
(rules and declarations) -
ending in semicolons. Called by RuleSetSource.EditRules to create rsOldRuleStrings and rsRuleStrings.
These lists are compared during RuleSet compilation. Rule objects are created to describe only those rules that have been changed.)

```
    (PROG (startPos stopPos ruleStrings)
          (SETQ stopPos 0)
          (SETQ ruleStrings (while (PROGN (SETQ startPos (ADD1 stopPos))
                                          (SETQ stopPos (STRPOS semicolon sourceRules startPos)))
                                collect (SUBSTRING sourceRules startPos stopPos)))
          (RETURN ruleStrings])
```

(OSSetCode

```
  [LAMBDA NIL                                                    (* mjs%: "11-OCT-82 15:42")
```

(* * Generate the Set Code for a one-shot rule. E.G. _@ (^task triedRule7) _T)

```
    (COND
       (rsTaskFlg                                                (* If tasking, set Task var.)
              (LIST '_@
                    '^task oneShotFlg T))
       (T                                                        (* If not tasking, set Rule var.)
          (LIST 'SETQ oneShotFlg T])
```

(OSTestCode

```
  [LAMBDA NIL                                                    (* mjs%: "11-OCT-82 13:26")
```

(* * Generate the Test Code for a one-shot rule. E.G., (NOT @ (^task triedRule7)))

```
    (COND
       (rsTaskFlg                                                (* Use Task Var if RuleSet to be used in a Task.)
              (LIST 'NOT (LIST '@ '^task oneShotFlg)))
       (T                                                        (* Use Rule Var RuleSet not to be used in a task.)
          (LIST 'NOT oneShotFlg])
```

(TraceLHSCodeGen

```
  [LAMBDA NIL                                                    (* RBGMartin "18-Feb-87 14:12")
```

(* * Generates varCode for tracing variables when the LHS of a rule is tested.)

```
    (PROG (varCode traceCode ruleCode code)
```

(* * Compute the parts of the trace code.)

```
          [COND
             (debugVars (SETQ varCode (CONS 'WRITETTY (for var in debugVars
                                                         join (LIST " " (CONCAT (UnParseTerm var)
                                                                                "="))
                                                              (CompileGetTerm var]
          (SETQ ruleCode (LIST '_ (LIST 'GetObjFromUID (KWOTE (UID ruleObject)))
                               'Print))
          (SETQ traceCode (LIST 'TraceLHS (KWOTE rsName)
                                (KWOTE ruleLabel)
                                ruleNumber))
```

(* * Splice the code together.)

```
          (SETQ code (LIST))
          [COND
             (varCode (SETQ code (CONS varCode code]
          [COND
             (ruleCode (SETQ code (CONS ruleCode code]
          [SETQ code (CONS 'PROGN (CONS '                        (* Rule Tracing Code)
                                        (CONS traceCode code]
          (RETURN code])
```

(TraceRHSCodeGen

```
  [LAMBDA NIL                                                    (* RBGMartin "18-Feb-87 14:12")
```

(* * Generates varCode for tracing variables when the RHS of a rule is satisfied.)

```
    (PROG (varCode traceCode ruleCode code)
```

(* * Compute the parts of the trace code.)

```
          [COND
             (debugVars (SETQ varCode (CONS 'WRITETTY (for var in debugVars
                                                         join (LIST " " (CONCAT (UnParseTerm var)
                                                                                "="))
                                                              (CompileGetTerm var]
          (SETQ ruleCode (LIST '_ (LIST 'GetObjFromUID (KWOTE (UID ruleObject)))
                               'Print))
          (SETQ traceCode (LIST 'TraceRHS (KWOTE rsName)
                                (KWOTE ruleLabel)
                                ruleNumber))
```

```
          (* * Splice the code together.)

          (SETQ code (LIST))
          [COND
             (varCode (SETQ code (CONS varCode code]
          [COND
             (ruleCode (SETQ code (CONS ruleCode code]
          [SETQ code (CONS 'PROGN (CONS '                    (* Rule Tracing Code)
                                        (CONS traceCode code]
          (RETURN code])

)

(RPAQQ ASSUMELISPFLG NIL)

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

FUNCTION INDEX

VARIABLE INDEX

PROPERTY INDEX