stichting
mathematisch
centrum

MC

DEPARTMENT OF COMPUTER SCIENCE IW 51/75 OCTOBER

H.J. BOOM & D. GRUNE

TEXTUAL MANAGEMENT IN AN ALGOL 68 COMPILER

5758.864

2e boerhaavestraat 49 amsterdam

Printed at the Mathematical Centre, 49, 2e Boerhaavestraat, Amsterdam.

AMS(MOS) Subject classification scheme (1970): 68A30

ACM-Computing Reviews-categories: 4.12

Textual management in an ALGOL 68 compiler

by

H.J. Boom & D.. Grune

ABSTRACT

A typical ALGOL 68 parser contains a number of well-studied and formally elegant algorithms, such as symbol-table management SLR(1) parsing, mode equivalencing, and coercion, together with a large amount of relatively ragged code to process the program text and use these algorithms at the proper time. A means of organizing this ragged code using affix transduction grammars is proposed.

KEY WORDS & PHRASES: *ALGOL 68 H, ALGOL 68, Affix grammar, Transduction grammar, compier, parsing, coercion.*

Ø. Preface

Throughout this article, the first person refers to the first author, H. J. Boom. D. Grune has done invaluable work in bringing the grammars to their present relatively readable form.

1. Introduction

In 1972, I began a one-man implementation of ALGOL 68. The compiler (called ALGOL 68 H) is now nearing completion. The severest constraint on the design of the compiler was that only one programmer (myself) was available for the project. This has forced several major design decisions.

The first major decision was to use the computer itself as an aid in ensuring correctness. To this end, an LR(k) parser generator was used to automate the writing of the parser. Furthermore, a secure higher-level language with reasonable control and data structures was chosen to write the compiler in. The implementation had to guarantee complete run-time and compile-time checking to prevent dangerous language violations leading to chaos. The only available convenient implementation meeting the requirements was ALGOL W. Although experience [BOOM 1] has shown that it has several drawbacks, the choice is not regretted.

The second major decision was to go for flexible processing and simplicity in all internal data structures. It occasionally happens that some unforeseen difficulty requires changes in previously debugged coding or in some data structure. It would simply not do if making changes in highly optimized data structures required complete redesign of large parts of the compiler. Furthermore, when the compiler was started, the language itself was still subject to revision.

The program was therefore represented internally, after parsing, as an explicit tree, linked together in the usual recursive way with pointers. This takes a lot of storage. All internal tables, including the symbol table and the mode table, are kept permanently in main storage. As a result, it is unlikely that the compiler will ever be able to compile real programs in a region much less than 4ØØ K bytes.

Nonetheless, I do not regret the decisions taken. The freedom of expression resulting from them has led to a great flexibility in designing compilation algorithms. Indeed, although their correctness had been informally proved, an intuitive feeling for some of the algorithms used was not obtained until after their programs had been coded and debugged.

Now, with hindsight, and with the experiences of reading the programs, debugging them, of looking at readable printouts of internal data structures, and of measuring the times spent in various parts of the compiler, it begins to become clear how to do much better. Essentially the same algorithms can be implemented using a quite different data structure for the parse tree. It is estimated that changing to a new data structure can save a factor of four to eight in storage consumption by the parse tree. There may also be a concomitant saving in CPU time, since a smaller data structure often needs fewer memory cycles for its examination. Furthermore, compilation can be done using only sequential scans of the new data structure for the parse tree. The resulting high predictability and locality of reference is a boon for software or hardware paging. It also makes it possible to keep intermediate texts on sequential media, such as magnetic tape.

Three other important data structures are the declaration table, the mode table, and the name table.

The declaration table can easily be broken up on a block-by-block basis. It can, in fact, be kept in the same file as the parse tree. Each block's definitions need be in main store only while the compiler is processing that block.

The name table contains information pertaining to the entire program. It contains the sequences of characters used in the program for representing tags, bold tags, and other indications and words. It is constructed during the lexical scan, and cannot be divided according to the block structure because the block structure is not yet known, and because declarations have not yet been parsed. The name table is, however, needed only during the lexical scan and for the production of readable error messages.

It might be possible to break the mode table up block by block, but the following example indicates that it may be difficult to decide to just which block any given mode belongs. It contains an applied occurrence of the tag "a" outside the range with its defining occurrence:

```
print(
    a of
      (mode m = struct(real a, b);
       m (3.5,4.2)
      )
    )
```

Elaboration of this unit should cause the value 3.5 to be printed.

2. ALGOL 68 H.

The present ALGOL 68 H compiler consists of five phases:

- Pre-scan.
- Parsing.
- Mode juggling.
- Coercion.
- Object code generation.

The pre-scan has two major functions. Its first function is breaking up the source text into a sequence of plain tags, bold tags, special characters, and denotations, while suppressing typographical display features. While doing so, it replaces indicators by pointers into the name table. Its second function is analysis of the bracket structure and detection of all mode, operation, and priority declarations. This information is used for proper classification of bold tags in the parsing phase.

The parsing phase builds a parse tree for the program. Although it would be pleasant if ALGOL 68 had a convenient LR(k) grammar, it does not. LR(k)-ness can be forced by ignoring certain important distinctions and permitting some constructions to associate in the wrong direction. ALGOL 68 H uses this approach, and it can therefore get away with an SLR(1) grammar. For example, no distinction is made between a bounds pair, a labelled unit, an actual parameter, and a trimmer. When, later in the parse, enough context has been accumulated to make such a distinction possible, a set of special routines is invoked to repair the parse tree, so that it will reflect the way the source program should have been parsed all along. This fixup operates top-down; the rest of the parser, driven by an LR(k) grammar, of course, operates bottom up. If this phase ever needs the mode of a mode indication, is simply uses some unique code associates with its defining occurrence instead.

The mode juggling phase finds the modes of all mode indications, builds a complete mode table, detects equivalent modes, chooses a canonical representative for each equivalence class, and ensures that all modes in the table have cross-references to the canonical representations instead of to other equivalent mode table entries. It is a remarkably straightforward piece of list processing, and appears to be extremely fast. It does not have to look at the source code or the parse tree.

The coercion phase identifies all indicators, determines all necessary coercions, and inserts the resulting information into the parse tree. In this phase, just as in the parsing phase, it is possible to distinguish some bottom-up and some top-down processing. Using the h-function [BOOM 2], a representative mode is computed for each construct bottom-up. This representative mode is not necessarily the true mode of the construct, but it is one that will work just as well for purposes of operator identification and coercion. For coercends, the representative mode is indeed its a priori mode. For a balance, the representative mode is the a priori mode of (one of) its strong coercend(s), if any. For certain constructs in the parse tree, this representative mode must then be adjusted in order to arrive at a target mode. For example, the primary of a slice must be of some REFETY ROWS MODE, and the mode of the operands of a formula must enable proper operator identification to take place. At this point, the syntactic position and its strength are used to attempt to coerce the representative mode until it is acceptable. If this succeeds, the target mode has been obtained. If it fails, error messages must be produced. After the target mode has been computed, the construct must be coerced to the proper mode. It is not sufficient to coerce the representative mode; Each of the coercends that have been balanced must now be coerced to its target mode (some changes in the target mode occur when coercing displays). The determination of the representative mode is a bottom up proces. Coercing the constructs to their proper target modes is a top-down process.

The object code generator has been, by far, the most difficult part of the compiler. Most of it has now been written, but some straightforward parts are not ready. It is at present too early to say anything of value about it in retrospect. I shall therefore refrain.

3. The new data structures.

The main improvement in the data structure for the parse tree comes from one simple discovery.

Only the parsing and coercion phases examine the program in its tree form (except for object code generation, which is another kettle of fish entirely). Each of these can be divided into two parts, a bottom-up part, and a top-down part. Furthermore, the top-down part need have no effect on the bottom-up part.

It is therefore possible to have all of these parts accept and produce Polish strings, in a strictly sequential fashion. Each part will consist of a single sequential scan, in reversed order, of the output from the previous scan. Although in theory, Polish postfix would be everywhere sufficient, it is found in practice that it simplifies coding to have each pass see an operator before its operand(s). After some passes, the source program is therefore left in prefix form; but after others, in postfix. This gives some advance warning that is of use with one or two kludges. One of the important properties of a good formalism is that it is possible to add kludges without destroying its structure.

The new design for the compiler therefore has seven phases:

	direction	input	output
- prescan	forward	source code	tokens, symbol table
- parse	forward	tokens	postfix, mode table
- parse back	backward	postfix	prefix
- juggle modes	-	mode table	mode table
- identify	forward	prefix, mode table	postfix
- coerce	backward	postfix, mode table	prefix
- code generation		prefix, mode table	object code

It should be emphasized that the grammars presented at the end of this paper, although they are sufficient to explain the algorithms involved to people, are not debugged, and likely contain many trivial errors. The algorithms themselves have been debugged in ALGOL 68 H, but the replacement of the explicit parse tree by Polish-like strings has probably introduced coding errors. My hope is that some implementor will take the method to heart in this new form. It would be irresponsible to abandon ALGOL 68 H, which is now a nearly finished ALGOL 68 compiler, in order to try out the new data structure. If one writes an infinite sequence of ever-improving half-compilers, one never gets a whole compiler finished. The algorithms are therefore being presented in their present incomplete form.

4. Affix transduction grammars.

The parse, parseback, identify, and coerce passes are described by transduction grammars in this paper. The parse and identify passes are described by LR(1) grammars, but the parseback and coerce phases require RL(1) grammars, because of the reversed processing direction. Except perhaps for the parse phase, it may well be easy to modify these grammars so that they are in fact LL(1) (or RR(1)).

It is felt that the easiest way to reach understanding of the algorithms presented here is to study the grammars and the transformations that the grammars induce on some typical source strings. To this end, we must explain the notion of a transduction grammar and the notation used for one in this paper.

A transduction grammar can be considered to be like an ordinary grammar, except that there is more than one kind of terminal symbol. In particular, there are "input symbols", and there are "output symbols".

In a normal grammar, one produces a string in its language by starting with a special non-terminal, the start symbol, and continually replacing a non-terminal on the left of a production by the string on the right side. If one ever succeeds in constructing a string exclusively of terminals, one has found a string of the language.

For transduction grammars, the rewriting process is quite similar, but the meaning of the string of terminals is different. Instead of defining a language, the transduction grammar defines a translation from one language to another. Each string of terminals produced as above specifies that some string of input symbols is to be translated to some string of output symbols. If only the input symbols of the produced string are considered (we imagine the output symbols to be erased), we have an input string. Similarly, if only the output symbols are considered, we have an output string. The transduction grammar specifies that the input string is in the input language and that its translation in the output language is the output string.

A transduction grammar is thus simultaneously an input specification, an output specification, and a specification of the relation between the two. If it is LR(k), it can be taken to specify the translation algorithm as well.

One further embellishment on transduction grammars is the attachment of parameters. To each symbol we can add some parameters. A typical parameter in the present application might be the mode of a construct. Formally, we specify that in any parse tree, it must be possible to attach parameters to the nonterminals in such a way that the various relationships specified in the grammar are satisfied. In practice, we simply compute the parameters in a straightforward right-to-left or left-to-right order while parsing. To assist in this, parameters are classified as value parameters and as result parameters. Value parameters are regarded as arising outside a branch of the parse tree and as being passed inwards; whereas result parameters involve information flow in the opposite direction. If a production is considered as a procedure, value parameters are passed to it by the caller and result parameters are passed back to the caller (just as in Algol W).

The result of adding parameters to a transduction grammar might be called an affix transduction grammar. Notice, however, that the affixes (parameters) are not quite the ones of KOSTER[1], and that the transduction rules are not quite the ones of LEWIS and STEARNS[1].

5. The notation used here.

In the grammar here presented, the distinction between input symbols and output symbols is not quite so strict. Sometimes a symbol is both an input symbol and an output symbol, in which case it is simply copied through. Furthermore, a symbol may be an input (only) symbol in one place, and an output (only) symbol in another place.

Productions are written in a manner reminiscent of Van Wijngaarden notation, with a nonterminal and a colon on the left, and a series of alternatives separated by semicolons on the right, followed by a period. The symbols within the alternatives are separated by commas.

Each nonterminal is written as an ALGOL 68 identifier, except that the positioning of spaces (but not their number) is significant, as in ordinary English. Thus, the following are distinct nonterminals:

```
alpha bet
alphabet
a lp hab et
alpha beta
pea soup
```

A terminal symbol is written as an ALGOL 68 quoted string. A terminal symbol may also be written as an underlined word, in which case it denotes itself.

Parameters are indicated by enclosing them in square brackets after the symbol. The value parameters are placed first, then an arrow, and then the result parameters. If there are no result parameters, the arrow itself may be omitted. Parameters are also omitted if they are unlikely to lead to the reader's enlightenment.

Comments also appear in the grammar, enclosed by square brackets. Distinguishing between comments and parameters is left to the readers intelligence.

At the beginning of each grammar there appears the word forward or backward, indicating the direction in which the scan proceeds.

The distinction between input symbols and output symbols is handled as follows. An output symbol which is not an input symbol is preceded by a slash (/). Input symbols which are not output symbols are enclosed in double angle brackets (<< and >>). In fact, from the symbols between angle brackets no output at all is produced, even if the symbol is a non-terminal and its own definition produces output. All other terminal symbols are both input and output symbols.

There are a number of symbols which act peculiarly:

/error: This one, which appears only as an output symbol, is not written to the normal output file; it instead causes an appropriate error message to appear on the file of error messages.

sr,er: These mark the start and end of a range, or of something which the compiler thinks might be a range. They always occur in pairs. If they occur as output symbols, the compiler should produce output indicating that a range is starting or ending, which range it is, and, if this is the last time the current phase will see the range, information enabling the next phase to find the definitions of this range in the symbol table. It is even possible to copy the appropriate part of the symbol table into the output text and erase it from random-access storage, since it will not be needed again until it is needed in the next scan. When one of these symbols is read, the action of te compiler is similar, but reversed.

/df: A definition of some symbol has just been encountered. Enter it into the symbol table for the current range.

mode: A symbol indicating an entry in the mode table. It may be either input or output, and may accordingly have a value or result parameter (the mode).

vardec, cast, dyad, and other bold symbols which do not appear in de Revised Report. These are terminal symbols used internally within the compiler. They should be considered as different from any similar indications which might appear in a program being compiled.

definition number: Each definition in the source program is assigned a unique code, by which the compiler can refer to it. Such a code might, for example, be an index into the symbol table. These codes are called "definition numbers". In a similar manner, we also have "symbol numbers".

6. An example

Let us examine an example:

```
(int a,
 proc(int) int b = skip;
 (a:b) (3) int c;
 (a:b) (3)
)
```

As we shall see shortly, this is an enormously complicated program.

The first pass performs lexical analysis. It finds no mode, operator, or priority declarations. It makes no significant changes in the program text (except that the lexical scan breaks it up into symbols).

After the parse pass (the second pass), the program appears as follows:

```
(sr int integral-mode modeind a integral-mode
      vardec,
   proc(int integral-mode modeind) int
         -integral-mode modeind
        procedure-with-integral-parameter-
           -yielding-integral-mode procp
        b procedure-with-integral-parameter-
              -yielding-integral-mode
       = skip iddec;
   (sr a : b er) serialclausepack l
        (sr 3 er) serialclausepack n
           int integral-mode modeind
              row-of-row-of-integral-mode
              c vardec
   (sr a : b er) serialclausepack l
        (sr 3 er) serialclausepack n closures
er) serialclausepack
```

After parseback, the program looks like this:

```
begin sr
vardec actual modeind int integral-mode
    df a reference-to-integral-mode noinit
          endvardec,
iddec
    formal procp(modeind int integral-mode)
        modeind int integral-mode
        procedure-integral-returning-integral-mode
        df b procedure-integral-returning-
              -integral-mode
        = skip;
vardec
    actual
        array (a : b)
            array (3)
                modeind int integral-mode
                row-of-integral-mode
            row-of-row-of-integral-mode
        df c reference-to-row-of-row-of-
              -integral-mode
        noinit
    endvardec;
call begin sr : df a : b er end
    number-one(sub 3)
er end
```

Of course, the symbols df do not actually form part of the output text; each is intercepted on its way out of the transduction, and instead, a definition is placed into the symbol table of the current block.

After identification, the program looks like this:

```
                                          begin sr
                                         series
                                      vardec
                int integral-mode modeind
                          a number-one
           reference-to-integral-mode tag
                                   noinit
                                   endvardec
                                     , iddec
procedure-with-integral-
-parameter-yielding-integral-mode fodecer
                       b number-two
procedure-with-integral-
 -parameter-yielding-integral-mode tag
                                        =
                                  skip
procedure-with-integral-parameter
                  -yielding-integral-mode
                                    endiddec
                                    ; vardec
                                     [
                      a
             number-one
reference-to-
        -integral-mode tag
reference-
       -to-integral-mode morf
                 integral-mode :
                            b
                   number-two
  procedure-with-integral-
      -parameter-yielding-
              -integral-mode tag
     procedure-with-integral-
         -parameter-yielding-
                 integral-mode morf
                error erroneous-mode ]
[ 3 integral-mode comorf integral-mode ]
         int integral-mode modeind
            row-of-integral-mode array
        row-of-row-of-integral-mode array
                                     c
                          number-three
               reference-to-row-of-
                -row-of-integral-mode tag
                                   noinit
                                   endvardec
                        ; begin sr series
       : a number-four label-mode tag :
                                b
                      number-two
     procedure-with-integral-
         -parameter-yielding-
                 -integral-mode tag
        procedure-with-integral-
-parameter-yielding-integral-mode morf
                         endseries er end
procedure-
  -with-integral-parameter-yielding
                            integral-mode
                               number-one
               (3 integral-mode comorf) call
                                      endseries
                                   er end voidmode
```

This version is difficult for people with western eyes to read, since it must be read from right-to-left instead of left-to-right. Identifiers have now been identified; one can see this by their definition-numbers. These definition-numbers are the same at the defining and applied occurrences. Notice the error message for the upper bound "b" in the array declarer; the mode of "b" could not be coerced to integral as required. Therefore, "erroneousmode" is used as the target mode in order to avoid further related messages; wherever the erroneous-mode appears it is assumed that an error message has already been produced concerning the problem.

Target modes are normally placed immediately to the right of the constructs that they coerce; in a few odd cases, such as with formulas, they are placed elsewhere. The next phase, coercion, will find these target modes, propagate them back towards their coercends and insert the appropriate coercion operators.

After the coercion phase (which operates from right to left), the program is left in prefix form, and looks like this:

```
begin sr
    series
        vardec modeind integral-mode int
            tag a number-one reference-to-integral-mode
            noinit
        endvardec,
        iddec
            tag b number-two
            = skip procedure-integral-returning-
                  -integral-mode
        endiddec;
        vardec
```

```
          array row-of-row-of-integral-mode
              [deref reference-to-integral-mode
                 tag a number-one reference-to-
                       -integral-mode
               :   erroneous integral-mode
                     tag b number-two procedure-
                           -integral-returning-
                           -integral-mode
              ]
              array row-of-integral-mode
                [  3  ]
                modeind integral-mode int
          tag c number-three reference-to-row-of-
                -row-of-integral-mode
          noinit
      endvardec;
      voiden
          call
              begin sr series
                : tag a number-four label-mode:
                tag b number-two procedure-integral-
                      -returning-integral-mode
              endseries er end
              number-one(  3  )
  endseries
er end
```

We may now see the coercion operators in their proper prefix places. Each coercion operator is supplied with mode(s) which may be useful in determining the operations it is to perform.

At this point all parsing and coercion has been performed, and the program text is ready to go to an object code generator.

[BOOM 1] H. J. Boom, Experience with the use of Algol W as SIL, Algol Bulletin 37.4.6, pp 63-67.

[BOOM 2] H. J. Boom, Note on balancing in ALGOL 68, Algol Bulletin 36.4.1, pp 17-24.

[KOSTER 1] C. H. A. Koster, Affix Grammars, in ALGOL 68 Implementation, the procedings of the IFIP Working Conference on ALGOL 68 implementation, Munich, July 20-24, 1970, North Holland, 1971.

[LEWIS and STEARNS] P. M. Lewis II and Stearns, Syntax-directed Transduction, JACM, vol. 15, no. 3, July 1968, pp. 465-488.

```
[ Phase 2: Parse ] "forward"

program:  closure;
     loop;
     /sr, labels, closure, /er;
     /sr, labels, loop, /er.

labels:  identifier, ":";
     labels, identifier, ":".

[The first phase has already checked that brackets of all kinds are
         properly matched.]
[The yielded n of a closure is used to count bound pairs in an array
         declarer.  It is zero for closures which cannot provide these
         bounds.]
closure[ -> 0]:  choice.
closure[ -> n]:
     "(", /sr, indexers or labellety unit list[ -> n], /er, ")", /coll;
     begin, /sr, indexers or labellety unit list[ -> n], /er, end,
         /coll;
     "[", /sr, indexers or labellety unit list[ -> n], /er, "]", /coll.
closure[ -> 1]:  "(", /sr, serial, /er, ")", /serialclausepack.
closure[ -> 0]:  begin, /sr, serial, /er, end /serialclausepack.
closure[ -> 1]:  "[", /sr, serial, /er, "]" /serialclausepack.
closure[ -> 1]:  "(", /sr, bounds, /er, ")" /boundspackorvacuum.
closure[ -> 0]:  begin end /vacuum.
closure[ -> 1]:  "[", /sr, bounds, /er, "]" /boundspack.
closure[ -> 0]:
     <<, par, >>, "(", /sr, indexers or labellety unit list, /er, ")",
         /par;
     <<, par, begin, >>, /"(", /sr, indexers or labellety unit list,
         <<end>>, /er, /")", /par.

choice:
     "(", /sr, choiceb, /er, <<, ")", >>, /fi;
     "(", /sr, choicei, /er, <<, ")", >>, /esac;
     "(", /sr, choiceu, /er, <<, ")", >>, /uesac;
     if, /sr, choiceb, /er, <<, ")", >>, /fi;
     <<case>>, /"(", /sr, choicei, /er, esac;
     <<case>>, /"(", /sr, choiceu, /er, <<esac>>, /uesac.

choiceb:
     serial, thenin, /sr, serial, /er, /"|", /sr, /skip, /er;
     serial, thenin, /sr, serial, /er, elseout, /sr, serial, /er;
     serial, thenin, /sr, serial, /er, elifouse, /"|", /sr, /"(", /sr,
         choiceb, /er, /fi, /er.

choicei:
     serial, thenin, /sr, indexers or labellety unit list, /er, /"|",
         /sr, /skip, /er;
     serial, thenin, /sr, indexers or labellety unit list, /er, elseout,
         /sr, serial, /er;
     serial, thenin, /sr, indexers or labellety unit list, /er,
         elifouse, /"|", /sr, /"(", /sr, choicei, /er, /esac, /er.

choiceu:
```

```
          serial, thenin, /sr, conf list, /er, /"|", /sr, /skip, /er;
          serial, thenin, /sr, conf list, /er, elseout, /sr, serial, /er;
          serial, thenin, /sr, conf list, /er, elifouse, /"|", /sr, /"(",
              /sr, choiceu, /er, /uesac, /er.

     thenin:
          <<, then, >>, /"|";
          "|";
          <<, in, >>, /"|".

     elseout:
          <<, else, >>, /"|";
          "|";
          <<, out, >>, /"|".

     elifouse:
          <<, elif, >>;
          <<, "|:", >>;
          <<, ouse, >>.

     conf list:
          conf;
          conf list, ",", conf.
     conf:  "(", /sr, ssvardec, ")", ":", unit, /er, /confrange;
          "(", /sr, declarer, ")", ":", unit, /er, /confnorange.

     serial:
          prelude, bounds or labellety unit;
          bounds or labellety unit.
     prelude:
          bounds or labellety unit, exit;
          bounds or labellety unit, ";";
          declaration, ";";
          prelude, bounds or labellety unit, exit;
          prelude, bounds or labellety unit, ";";
          prelude, declaration, ";".

     indexers or labellety unit list[ -> 2]:
          indexer, ",", indexer;
          indexer, ",", bounds or labellety unit;
          bounds or labellety unit, ",", indexer;
          bounds or labellety unit, ",", bounds or labellety unit.
     indexer or labellety unit list[ -> n+1]:
          indexer or labellety unit list[ -> n], ",", indexer;
          indexer or labellety unit list[ -> n], ",", bounds or labellety
              unit.

     bounds or labellety unit:
          unit;
          bounds or labellety unit, ":", unit.

     indexer:
          bounds, "@", unit;
          bounds or labellety unit, "@", unit;
          bounds.
```

```
bounds:  bounds, ":", unit;
     /missing;
     bounds, ":", /missing;
     bounds or labellety unit, ":", /missing.
     ['bounds' always contain at least one missing unit.]

declaration:  popdec;
     popdec, ",", declaration;
     opdec;
     opdec, ",", declaration;
     priodec;
     priodec, ",", declaration;
     iddec;
     iddec, ",", declaration;
     ssvardec;
     ssvardec, ",", declaration;
     svardec;
     svardec, ",", declaration;
     vardec;
     vardec, ",", declaration;
     modec;
     modec, ",", declaration;
     procdec;
     procdec, ",", declaration;
     procvardec;
     procvardec, ",", declaration.

popdec:  <<, op, >>, operator, "=", routine text, /popdec;
     popdec, ",", operator, "=", routine text, /popdec.

opdec[ -> mode]:  <<op>>, virtual plan[ -> mode], operator, /mode[mode
         -> ], "=", unit, /opdec;
     opdec[ -> mode], ",", operator, /mode[mode -> ], "=", unit,
         /opdecx.

priodec:  <<, prio, >>, operator, "=", unit, /priodec;
     priodec, ",", operator, "=", unit, /priodec.

operator:
     "=";
     other operator.

iddec[ -> mode]:  declarer[ -> mode], identifier, /mode[mode -> ], "=",
         unit, /iddec;
     iddec[ -> mode], ",", identifier, /mode[mode -> ], "=", unit,
         /iddec.

vardec[ -> mode]:  leap, declarer[ -> mode], identifier, /mode[mode ->
         ], /vardec;
     leap, declarer,[ -> mode], identifier, /mode[mode -> ], ":=", unit,
         /vardeci;
     declarer,[ -> mode], identifier, /mode[mode -> ], ":=", unit,
         /vardeci;
     vardec[ -> mode], ",", identifier, /mode[mode -> ], /vardecx;
     vardec[ -> mode], ",", identifier, /mode[mode -> ], ":=", unit,
         /vardecix;
```

```
    svardec[ -> mode], ",", identifier, /mode[mode -> ], ":=", unit,
        /vardecix;
    ssvardec[ -> mode], ",", identifier, /mode[mode -> ], ":=", unit,
        /vardecix.

svardec[ -> mode]:
    ssvardec[ -> mode], ",", identifier, /mode[mode -> ], /vardecx;
    svardec[ -> mode], ",", identifier, /mode[mode -> ], /vardecx.

ssvardec[ -> mode]:  declarer[ -> mode], identifier, /mode[mode -> ],
        /vardec.

modec:  <<mode>>, declarer[ -> m], "=", declarer[ -> n], /equate[m,n ->
        ], /modec;
    modec, ",", declarer[ -> m], "=", declarer[ -> n], /equate[m,n ->
        ], /modec.

procdec:  proc, identifier, "=", routine text, /procdec;
    procdec, ",", identifier, "=", routine text, /procdec.

procvardec:
    proc, identifier, ":=", routine text, /procvardec;
    leap, proc, identifier, ":=", routine text, /procvardec;
    procvardec, ",", identifier, ":=", routine text, /procvardec.

declarer[ -> M]:  [ The computation of the mode M of a declarer is
        straightforward, and is not shown.  ]
    ordinary declarer;
    array declarer.
ordinary declarer:
    mode indication, /mode[M -> ], /modeind;
    <<, ref, >>, declarer, /mode[M -> ], /ref;
    proc, declarer, /mode[M -> ], /proc;
    <<, flex, >>, declarer, /mode[M -> ], /flex;
    <<, struct, >>, "(", fields, ")", /mode[M -> ], /struct;
    <<, union, >>, "(", declarer list, ")", /mode[M -> ], /union;
    proc, virtual plan, /mode[M -> ], /procp.
array declarer:
    closure[ -> n], /1, ordinary declarer, /mode[M -> ], /array,
        check[The number of rows, N, must be greater than zero.  ];
    closure[ -> n], /1, array subdeclarer, /mode[M -> ], /array,
        check[The number of rows, N, must be greater than zero.  ].
array subdeclarer:
    closure[ -> n], /n, ordinary declarer, /mode[M -> ], /array,
        check[The number of rows, N, must be greater than zero.  ];
    closure[ -> n], /n, array subdeclarer, /mode[M -> ], /array,
        check[The number of rows, N, must be greater than zero.  ].
    [The number of rows of a closure is zero for certain constructions,
        such as choice clauses, which cannot be used as bounds.  There
        should be a better way to impose this condition.]
fields:  declarer, identifier list;
    fields, ",", declarer, identifier list.
identifier list:  identifier;
    identifier list, ",", identifier.
declarer list:  declarer;
    declarer list, ",", declarer.
```

```
virtual plan:  "(", declarer list, ")", declarer.

closures:
     closure, /l, closurex;
     closure, /l.
closurex:  closure, /n, closurex;
     closure, /n.

unit:
     tertiary;
     tertiary, <<, ":=", >>, unit, /":=";
     tertiary, <<, is, >>, tertiary, /is;
     tertiary, <<, isnt, >>, tertiary, /isnt;
     routine text;
     loop;
     skip;
     goto, identifier, /goto.
goto:  <<, go, to, >>;
     <<, goto, >>.
     [ ???  and what about comments betweem go and to ???  ]

routine text:  "(", /sr, formal parameters proper, ")", declarer, ":",
         unit, /er, /endtextp.
     "(", /sr, ssvardec, ")", declarer, ":", unit, /er, /endtextp;
     declarer, ":", unit, /endtext.
formal parameters proper:
     svardec;
     svardec, ",", formal parameters proper;
     ssvardec, ",", formal parameters proper.

loop:  <<, forety, >>, fromety, byety, toety, /sr, /forety, /sr,
         whilety, do, /sr, serial, /er, /er, /er, od.
forety:  /for, /missing;
     for, identifier;
fromety:  /from, /missing;
     from, unit.
byety:  /by, /missing;
     by, unit.
toety:
     /to, /missing;
     to, unit.
whilety:  /while, /missing;
     while, serial.

tertiary:
     nil;
     formula;
formula:  priority 1 operand.
priority 1 operand:
     priority 1 operand, priority 1 operator, priority 2 operand, /dyad;
     priority 2 operand.
priority 2 operand:
     priority 2 operand, priority 2 operator, priority 3 operand, /dyad;
     priority 3 operand.
priority 3 operand:
     priority 3 operand, priority 3 operator, priority 4 operand, /dyad;
     priority 4 operand.
priority 4 operand:
     priority 4 operand, priority 4 operator, priority 5 operand, /dyad;
     priority 5 operand.
priority 5 operand:
     priority 5 operand, priority 5 operator, priority 6 operand, /dyad;
     priority 6 operand.
priority 6 operand:
     priority 6 operand, priority 6 operator, priority 7 operand, /dyad;
     priority 7 operand.
priority 7 operand:
     priority 7 operand, priority 7 operator, priority 8 operand, /dyad;
     priority 8 operand.
priority 8 operand:
     priority 8 operand, priority 8 operator, priority 9 operand, /dyad;
     priority 9 operand.
priority 9 operand:
     priority 9 operand, priority 9 operator, monadic operand, /dyad;
     monadic operand.
monadic operand:
     monadic operator, monadic operand, /monad;
     secondary.

secondary:  leap, declarer[ -> m], /mode[enref(m) -> ], /gen;
     identifier, <<, of, >>, secondary, /of;
     primary, closurex, /pclosures;
     closures, /closures;
     declarer, closures, /cast;
     declarer, loop, /loopcast;
     primary.

leap:
     heap;
     local.

primary:  denotation;
     identifier;
     "$", collection list, "$".
```

```
[ Phase 3: Parseback ]
"backward"


program:  unit;
     sr, labels, unit, er.

unit:  unit not identifier;
     identifier;
     /error, missing.

unety:  unit not identifier;
     identifier;
     missing.

unit not missing:  unit not identifier;
     identifier.

unit not identifier:
     /par, begin, <<, sr, >>, unit list, <<, er, >>, end, <<, par, >>;
     /coll, begin, <<, sr, >>, unit list, <<, er, >>, end, <<, coll, >>;
     begin, sr, serial, er, end, <<, serialclausepack, >>;
     /if, <<, "(", >>, sr, series, /then, <<, "|", >>, sr, serial, er,
         /else, <<, "|", >>, sr, serial, er, er, fi;
     /case, <<, "(", >>, sr, series, /in, <<"|", sr, >>, unit list, <<,
         er, >>, /out, <<, "|", >>, sr, serial, er, er, esac;
     /ucase, <<, "(", >>, sr, series, /uin, <<, "|", sr, >>, confs, <<,
         er, >>, /uout, <<, "|", >>, sr, serial, er, er, uesac;
     /vacuum, begin, end, <<, vacuum, >>;
     /vacuum, begin, <<, sr, missing indexer, er, >>, end, <<,
         boundspackorvacuum, >>;

     loop;
     routine text;

     /":=", unit, unit, <<, ":=", >>;
     /is, unit, unit, <<, is, >>;
     /isnt, unit, unit, <<, isnt, >>;
     /dyad, unit, operator, unit, <<, dyad, >>;
     /monad, operator, unit, <<, monad, >>;
     /gen, leap, actual declarer, mode, <<, gen, >>;
     /of, identifier, unit, <<, of, >>;
     denotation;
     identifier;
     skip;
     nil;
     /goto, identifier, <<, goto, >>;
     "$", collection list, "$".

unit list:  unit as if bounds, ",", unit list;
     unit as if bounds.
unit as if bounds:  unit;
     /error, "@", unit;
     /error, ":", unit as if bounds;
     /error, unety, unit as if bounds.
```

```
series option:
     series;
     missing.

missing indexer:  missing;
     /error, unit not missing;
     "@", /error, missing indexer.

loop:
     from, unety, by, unety, to, unety, sr, for, /df, unety, sr, while,
          series option, do, sr, serial, er, er, er, od.

conf list:
     conf;
     conf, ",", conf list.
conf:
     /confrange, <<, "(", >>, sr, df, declarer, identifier, <<, vardec,
          ")", ":", >>, unit, er, <<, confrange, >>;
     /confnorange, <<, "(", sr, >>, declarer, <<, ")", ":", >>, unit,
          <<, er, confnorange, >>.

routine text:
     /textp, /mode[m -> ], "(", sr, formal parameters, ")", formal
          declarer, ":", unit, er, endtextp;
     /text, /mode[m -> ], sr, formal declarer, ":", unit, er, endtext.

formal parameter list; formal parameter;
     formal parameter, ",", formal parameter list.
formal parameter:  formal declarer, /df, identifier, mode, <<, vardec,
          >>;
     /df, identifier, mode, <<, vardecx, >>.

[ serial clauses and series ]

serial:
     [ A serial may contain labels.  ]
     declaration prelude, labellety, labellety postlude;
     labellety postlude.
labellety postlude:
     unit;
     unit, exit, labels, labellety postlude;
     unit, /error, exit, labellety postlude;
     unit, ";", labellety, labellety postlude;
     unit, /";", /error, <<, "@", >>, labellety postlude.
labellety:
     ;
     labels.

series:
     [ A series may not contain labels.  ]
     declaration prelude, labellety lament, units postlude;
     units postlude.
units postlude:
     unit;
     unit, error, exit, labellety lament, units postlude;
     unit, ";", labellety lament, units postlude;
     unit, /";", /error, <<, "@", >>, units postlude.
labellety lament:
     ;
     /error, labels.

labels:
     /":", /df, label, ":", labels;
     /":", /df, label, ":".
label:  /error, unit not identifier;
     /df, identifier;
     /error, /identifier, missing.

[ Declarations and declarers.  ]

declaration prelude:  declaration, ";";
     labellety lament, declaration prelude;
     unit, /";", /error, <<, "@", >>, declaration prelude;
     declaration, ";", declaration prelude.

declaration:  single declaration;
     single declaration, ",", declaration.

single declaration:
     /op, /missingdeclarer, /df, operator, /mode[m -> ], "=", routine
         text[ -> m], <<, popdec, >>;
     /op, /formal, /procp, virtual plan, /mode[m -> ], /df, operator,
         mode, "=", unit, <<, opdec, >>;
     <<, prio option, operator, "=", unit, >>, priodec;
     /iddec, formal declarer, /df, identifier, mode, "=", unit, <<,
         iddec, >>;
     /iddec, /missingdeclarer, /df, identifier, mode, "=", unit, <<,
         iddec, >>;
     /modec, /df, mode indication, "=", actual declarer, <<, modec, >>;
     /iddec, /missingdeclarer, proc option, /df, identifier, /mode[m ->
         ], "=", routine text[ -> m], <<, procdec, >>;
     /vardec, actual declarer, vardefs, /endvardec;
     /vardec, /missingdeclarer, proc option, /df, identifier, /mode[ref
         m -> ], ":=", routine text[ -> m], /endvardec, <<, procvardec,
         >>.

prio option:  ;
     <<, prio, >>.

proc option:  ;
     <<, proc, >>.

vardefs:  firstvardef;
     vardefs, ",", vardef.
firstvardef:  /df, identifier, /mode[ref m], <<, mode[ -> m], >>,
         /noinit, <<, vardec, >>;
     /df, identifier, /mode[ref m -> ], <<, mode[ -> m], >>, ":=", unit,
         <<, vardeci, >>.
vardef:  /df, identifier, /mode[ref m -> ], <<, mode[ -> m], >>,
         /noinit, <<, vardecx, >>;
     /df, identifier, /mode[ref m -> ], <<, mode[ -> m], >>, ":=", unit,
         <<, vardecix, >>.
```

```
actual declarer[ -> m]:  /actual, declarer.
virtual declarer[ -> m]:  /virtual, declarer.
formal declarer[ -> m]:  /formal, declarer.
[ the identification scan will check that every declarer is indeed of
     the proper VICTALity.  ]

declarer:
     /ref, declarer, mode, <<, ref, >>;
     proc, declarer, mode, <<, proc, >>;
     /flex, declarer, mode, <<, flex, >>;
     /struct, "(", fields, ")", mode, <<, struct, >>;
     /union, "(", declarer list, ")", mode, <<, union, >>;
     /array, victal bounds pack, declarer, mode, <<, array, >>;
     /procp, <<, proc, >>, virtual plan, mode, <<, procp, >>;
     /modeind, mode indication, mode, <<, modeind, >>.

fields:
     declarer, identifier list;
     declarer, identifier list, ",", fields.

declarer list:
     declarer;
     declarer, ",", declarer list.

victal bounds pack:
     sub, <<, sr, >>, victal bounds, <<, er, >>, bus, closure trailer,
         nl.

victal bounds:  unety, ":", unety;
     unety;
     /error, wrong victal bounds.
wrong victal bounds:
     <<, unety, "@", >>, unit;
     <<, unety, ":", >>, unety, ":", unety;
     <<, unety, ":", >>, wrong victal bounds.

closure trailer:
     <<, coll, >>;
     <<, serialclausepack, >>;
     <<, boundspack, >>;
     <<, boundspackorvacuum, >>.

nl:
     <<, n, >>;
     <<, l, >>.

virtual plan:  "(", declarer list, ")", declarer.
```

[Analysis of closures]

[Notice that no distinction is made between calls and slices at this point. If it were done, distinguishing those slices that have "@" or ":" in them might give the next scan advance information to slightly improve error recovery.]

```
unit not identifier:  pclosures, <<, pclosures, >>;
     closures, <<, n, closures, >>;
     cast, <<, cast, >>;
     /cast, declarer, loop, <<, loopcast, >>.

pclosures:
     /call, pclosures, parameters pack, <<, n, >>;
     /call, unit, parameters pack, <<, n, >>.

closures:
     /call, closures, <<, n, >>, parameters pack;
     /call, unit, <<, 1, >>, parameters pack.

cast:  /call, cast, parameters pack, <<, n, >>;
     /cast, declarer, closure, <<, 1, >>.
parameters pack:
     /number[n -> ], sub, <<, sr, >>, parameters[ -> n], <<, er, >>,
         bus, closure trailer.

parameters[ -> n+1]:  trimscript, ",", parameters[n].
parameters[ -> 1]:  trimscript.

trimscript:  /sub, subscript;
     /trim, trimmer;
     missing.
subscript:  unit not missing.
trimmer:  unety, ":", unety;
     /missing, "@", unit;
     unety, ":", unety, "@", unit;
     /error, <<, unety, ":", >>, trimmer.

sub:  "[";
     "(";
     /"[", error, <<begin, >>.
bus:  "]";
     ")";
     /"]", error, <<, end, >>.
begin:  /begin, <<, "(", >>;
     begin;
     /begin, /error, <<, "[", >>.
end:  /end, <<, ")", ">>";
     end;
     /end, /error, <<, "]", >>.
```

```
[ Phase 5: Identify ] "forward"
program:  sr, labels, unit[ -> r], force[r, strong, void], er;
     unit[ -> r], force[r, strong, void].

unit[ -> r]:  coercend n i[ -> r];
     applied identifier[ -> r];
     other unit[ -> r].
unit[ -> "missing"]:  missing.

coercend n i[ -> r]:
     morf n i[ -> r], /mode[r -> ], /morf;
     comorf n i[ -> r], /mode[r -> ], /comorf.

unit n i[ -> r]:
     coercend n i[ -> r];
     other unit[ -> r].
unit n i[ -> "missing"]:  missing.

unit not missing[ -> r]:  coercend n i[ -> r];
     applied identifier[ -> r];
     other unit[ -> r].

missing unit:  missing;
     unit not missing, /error.

applied identifier[ -> r]:
     identifier[ -> r], /mode[r -> ], /morf.

identifier[ -> mode]:  tag[ -> symbol number],
     identify tag[symbol number -> definition number, mode], /definition
         number[definition number -> ], /mode[mode -> ], /tag.
     [ A single tag can be a jump, but in that case it doesnt matter
         whether it is classified as a morf or a comorf -- the mode,
         "label", is a special case.  ]
     [ The tag itself is kept only because it may be useful in error
         messages.  Of course, if the tag or a reference to it were
         placed in the definition table, it would not have to be kept in
         the text.  However, one would then have to do something about
         undeclared tags.  ]
identify tag[symbol number -> definition number]:
     [ Find the currently active definition of the symbol ′symbol
         number′, if any, and yield its definition number.  If no active
         definition exists, complain to the programmer, and do something
         sensible.  ]

other unit[ -> void]:
     from, integral unety, by, integral unety, to, integral unety, sr,
         for, identifier option, sr, while, boolean series option, do,
         sr, series, er, er, er, od.
identifier option:
     missing;
     identifier.

boolean series option:  missing;
     series[ -> r], force[r, meek, boolean].
```

```
integral unety:  missing;
    unit[ -> r], force[r, meek, integral].

other unit[ -> hip]:
    par, <<, begin, >>, unit list, <<, end, >>, /parend;
    coll, <<, begin, >>, unit list, <<, end, >>, /collend;
    vacuum, <<, begin, end, >>.
unit list:  unit[ -> r];
    unit list, ",", unit[ -> r].

other unit[ -> r]:
    begin, sr, series[ -> r], er, end.

other unit[ -> balance(r1, r2)]:
    if, sr, series[ -> rc], force[rc, meek, boolean], then, sr, series[
        -> r1], er, else, sr, series[ -> r2], er, er, fi;
    case, sr, series[ -> rc], force[rc, meek, integral], in, balanced
        unit list[ -> r1], out, sr, series[ -> r2], er, er, esac;
    ucase, sr, series[ -> rc], force[rc, meek, united], in, confronter
        list[ -> r1], out, sr, series[ -> r2], er, er, esac.

balanced unit list[ -> r]:  unit[ -> r].
balanced unit list[ -> balance(r1, r2)]:
    balanced unit list[ -> r1], ",", unit[ -> r2].

confronter list[ -> r]:  confronter[ -> r].
confronter list[ -> balance(r1, r2)]:
    confronter list[ -> r1], ",", confronter[ -> r2].
confronter[ -> r]:
    confrange, sr, formal declarer, identifier, unit[ -> r], er,
        /endconfrange;
    confnorange, declarer, unit[ -> r], /endconfnorange.

series[ -> r]:  /series, unit[ -> r], /endseries.
series[ -> balance(r1, r2)]:
    /series, prelude[ -> r1], unit[ -> r2], /endseries.
prelude[ -> "missing"]:
    unit, ";";
    ":", identifier, ":";
    declaration, ";".
prelude[ -> r]:  prelude[ -> r], unit, ";";
    prelude[ -> r], ":", identifier, ":";
    prelude[ -> r], declaration, ";";
    unit[ -> r], exit.
prelude[ -> balance(r1, r2)]:
    prelude[ -> r1], unit[ -> r2], exit.

[ calls ]

morf n i[ -> pp(s, p)]:
    <<, call, >>, unit, force[f, meek, procparams -> s], number[ -> n],
        check[n = nparams(s)], open, parameters[1, s -> p], close,
        /call.

    [This production, alas, requires that the affixes of 'force' be
        used to distinguish calls from slices.  This is one of the less
        serious aspects of the square bracket problem.  To get this
        through the average lr(k) grammar checker, however, it will be
        necessary to introduce a kludge.  Perhaps the call of 'force'
        could be made to inject an appropriate symbol into the input
        stream which the parser could then recognize in its lookahead.
        ]

     [ Explanation of pp(s, p):
       s is a mode for procedures with parameters, and
       p is a collection of integers.
       construct the new mode pp(s, p) from s by keeping the i-th
         parameter mode of 's' only if 'i' is in 'p'.  If partial
         parameterization is not to be implemented, this function should
         be replaced by another, which returns the mode returned by 's',
         and complains if 'p' does not specify deletion of all the
         parameter modes.  ]

parameters[i, s -> p]:
     parameter[i-th parameter mode of s -> ], ",", parameters[i+1, s ->
        p].
parameters[i, s -> p union {i}]:
     missing, ",", parameters[i+1, s -> p].
     [ "union" denotes the set union operator.  ]
parameters[i, s -> {}]:
     parameter[i-th submode of s -> ].
parameters[i, s -> {i}] :  missing.
parameter[m -> ]:
     <<, sub, >>, unit[ -> r], force[r, strong, m];
     <<, trim, >>, /error, unit[ -> r], force[r, strong, m], <<,
        improper parameter tail, >>.
improper parameter tail:
     ;
     ":", unit, improper parameter tail;
     "@", unit, improper parameter tail.

[slices ]

morf n i[ -> derow(s, p)]:
     <<, call, >>, unit[ -> r], force[r, meek, refetyrow -> s], number[
        -> n], check[n = dimensions(s)], sub, trimscripts[ -> p], bus,
        /slice.

     [this production partakes of the affix problem mentioned for
        calls.]

     [Explanation of derow(s, p):
       s is a mode, "refety ROWS of MODE", and
       p is an integer.
       construct a new mode, yielded by derow, by deleting p occurrences
         of "row" from the "ROWS" in s.
       Delete the "of" if necessary.  ]

trimscripts[ -> p+1]:  subscript, ",", trimscripts[ -> p].
trimscripts[ -> p] :
     trimmer, ",", trimscripts[ -> p].
subscript:  sub, bound.
```

```
trimmer: missing;
     trim, boundety, ":", boundety;
     trim, boundety, ":", boundety, "@", bound;
     trim, missing, "@", bound.
boundety: missing;
     bound.
bound: unit not missing[ -> r], force[r, meek, integral].

morf n i[ -> yield]:
     <<, dyad, >>, unit[ -> r1], operator[ -> symbol number], unit[ ->
        r2],
       identify dyadic operator[r1, symbol number, r2 -> definition
        number, t1, t2, yield],
       /definition number[definition number -> ],
       /mode[t1 -> ], /mode[t2 -> ], /dyad.
identify dyadic operator[r1, symbol, r2 -> definition, t1, t2, yield]:
     [ find the definition, whose definition number is 'definition', of
        the dyadic operator 'symbol', such that r1 (r2) can be firmly
        coerced to its argument mode t1 (t2), and such that the
        operator returns values of the mode 'yield'.  If there is no
        such definition currently active, produce an error message and
        some sensible action].

morf n i[ -> yield]:
     <<, monad, >>, operator[symbol number], unit[ -> r],
       identify monadic operator[symbol number, r -> definition number,
        t, yield],
       /definition number[definition number -> ],
       /mode[t -> ], /monad.
identify monadic operator[symbol, r1, -> definition, t1, yield]:
     [ find the definition, whose definition number is 'definition', of
        the monadic operator 'symbol', such that r1 can be firmly
        coerced to its argument mode t1, and such that the operator
        returns values of the mode 'yield'.  If there is no such
        definition currently active, produce an error message and some
        sensible action].

morf n i[ -> symbol number of s]:
     <<, of, >>, tag[ -> symbol number], unit[ -> r], force[r, weak,
        refetyrowsetystruct -> s], /of.
     [Explanation of of operator:
       "symbol number" is a number of a field selector.
       "s" is a mode of the form "REFETY ROWWTY STRUCT".
       The proper field mode, F, is selected from the STRUCT, and then
        the of operator yields "REFETY ROWSETY F".  ]

morf n i[ -> r]:
     text, mode[ -> r], <<, sr, declarer, ":", >>, unit, <<, er, >>,
        endtext;
     textp, mode[ -> r], "(", sr, formal parameter list, ")", <<,
        declarer[ -> y], ":", >>, unit[ -> r], force[r, strong, y], er,
        endtextp.
formal parameter list:
     formal parameter;
     formal parameter list, ",", formal parameter.
formal parameter:
     <<, declarer, >>, identifier, mode;
     identifier, mode.

comorf[ -> s]:
     <<, ":=", >>, unit[ -> rd], force[rd, soft, refmode -> s], unit[ ->
         rs], force[rs, strong, deref(s)], /":=".

comorf[ -> boolean]:
     <<, is, >>, tertiary[ -> r1], tertiary[ -> r2], force[balance(r1,
         r2), soft, refmode], /is;
     <<, isnt, >>, tertiary[ -> r1], tertiary[ -> r2], force[balance(r1,
         r2), soft, refmode], /isnt.

tertiary[ -> r]:  unit n i[ -> r];
     applied identifier[ -> r], check[ r /= "label" -> ].

comorf[ -> m]:  <<, gen, >>, leap, declarer, mode[ -> m], /gen.

comorf[ -> t]:  <<, cast, declarer[ -> t], >>, unit[ -> r], force[r,
         strong, t], /cast.

comorf[ -> r]:  denoter[ -> r].

comorf[ -> "format"]:  "$", collection list, "$".

other unit[ -> nihil]:  nil.
other unit[ -> hip]:  skip.
other unit[ -> "jump"]:
     <<, goto, >>, identifier[ -> r], check[ r = "label" ], /goto.

declaration:  single declaration;
     declaration, ",", single declaration.

single declaration:
     <<, op, >>, /iddec, declarer, /optag, operator[ -> symbol number],
         <<, mode[ -> m], >>,
       identify defining operator[symbol number, m -> definition
         number],
       /definition number[definition number],
       /mode[m -> ], "=", unit[ -> r], force[r, strong, m], /endiddec;
     iddec, declarer, indicator, mode[ -> m], "=", unit[ -> r], force[r,
         strong, m], /endiddec;
     modec, mode indication, "=", declarer, /endmodec;
     vardec, declarer, vardef list, endvardec;
     [Because of the funny semantics of joined variable definitions,
         they are still joined here.  others have been broken apart.  ]
     priodec.

vardef list:  vardef;
     vardef list, ",", vardef.
vardef:
     identifier, <<, mode[ -> m], >>, initialization option[m -> ].

initialization option[m -> ]:
     noinit;
     ":=", unit[ -> r], force[r, strong, m].
```

```
declarer[ -> m]:
     <<, actual, >>, actual declarer[ -> m];
     <<, virtual, >>, virtual declarer[ -> m];
     <<, formal, >>, formal declarer [ -> m].

[The mode juggler must catch flex flex.  ]

victal declarer[ -> m]:
     <<, ref, virtual declarer, >>, mode[ -> m];
     <<, proc, formal declarer, >>, mode[ -> m];
     <<, procp, "(", formal declarer list, ")", formal declarer, >>,
         mode[ -> m];
     <<, union, "(", formal declarer list, ")", >>, mode[ -> m].

actual declarer:  victal declarer, /acdecer;
     <<, struct, >>, "(", actual fields, ")", mode[ -> m], /struct;
     <<, array, >>, sub, actual bounds, bus, actual declarer, mode[ ->
         m], /array;
     <<, flex, >>, actual declarer, mode[ -> m], /flex;
     <<, modeind, >>, mode indication, mode[ -> m], /modeind.

virtual declarer[ -> m]:
     victal declarer, /videcer;
     <<, struct, "(", virtual fields, ")", >>, mode[ -> m], /videcer;
     <<, array, sub, virtual bounds, bus, virtual declarer, >>, mode[ ->
         m], /videcer;
     <<, flex, virtual declarer, >>, mode[ -> m], /videcer;
     <<, modeind, mode indication, >>, mode[ -> m], /videcer.

formal declarer:
     victal declarer, /fodecer;
     <<, struct, "(", formal fields, ")", >>, mode[ -> m], /fodecer;
     <<, array, sub, formal bounds, bus, formal declarer, >>, mode[ ->
         m], /fodecer;
     <<, flex, /error, formal declarer, >>, mode[ -> m], /fodecer;
     <<, modeind, mode indication, >>, mode[ -> m], /fodecer.

actual fields:  actual declarer, tag;
     actual fields, ",", actual declarer, tag;
     actual fields, ",", tag.
virtual fields:  virtual declarer, tag;
     virtual fields, ",", virtual declarer, tag;
     virtual fields, ",", tag.
formal fields:  formal declarer, tag;
     formal fields, ",", formal declarer, tag;
     formal fields, ",", tag.

formal declarer list:  formal declarer;
     formal declarer list, ",", formal declarer.

actual bounds:  actual bound pair;
     actual bounds, ",", actual bound pair.
actual bound pair:  actual bound, ":", actual bound;
     actual bound.
actual bound:  unit[ -> r], force[r, meek, integral].
virtual bounds:  virmal bounds.
```

```
formal bounds: virmal bounds
virmal bounds: virmal bound pair;
     virmal bounds, ",", virmal bound pair.
virmal bound pair: missing unit;
     missing unit, ":", missing unit.

force[apriori, sort, target -> result]:
     coerce[apriori, sort, target -> result], /mode[result -> ].

coerce[apriori, SORT, target -> result]:
     [determine whether there is a mode result, which:
       - can be SORTly coerced from apriori, and
       - is the mode, or is in the mode class, target.
     If not, except in the special case of the kludge for distinguishing
          slices and calls, produce an appropriate error message.
     Valid values of apriori are:
       - any mode, or void,
       - the special modes hip, nihil, jump, undefined, or label.
     Valid values of target are:
       - any mode, or void,
       - refetyrowsetystruct,
       - refetyrow,
       - ref,
       - procparams.
     ´sort´ is strong, firm, meek, weak, or soft.] .

sub:
     <<, "(", >>, /"[";
     "[".
bus:
     <<, ")", >>, "]";
     "]".
open:
     "(";
     <<, "[", >>, /error, /"(".
close:
     ")";
     <<, "]", >>, /error, /")".
begin:
     <<, "(", >>, begin;
     begin.
end:
     /end, <<, ")", >>;
     end.
```

```
[ Phase 6: Coerce ] "backward"

program:
     sr, labels, unit[void, "transety"], er;
     unit[void, "transety"].

labels:
     ":", defining identifier, ":";
     labels, ":", defining identifier, ":".

unety[target, transety -> ]:
     unit[target, transety -> ];
     missing.

unit[mode, t -> ]:
     if, sr, unit, then, sr, unit[mode, t], er, else, sr, unit[mode, t
         -> ], er, er, fi;
     case, sr, unit, in, unit list[mode, t -> ], out, sr, unit[mode, t
         -> ], er, er, esac;
     ucase, sr, unit, uin, conformer list[mode, t -> ], uout, sr,
         unit[mode, t -> ], er, er, esac;
     begin, sr, unit[mode, t -> ], er, end;
     par, check[mode = "void"], collateral list[1, mode, t -> ], parend;

unit[target, transety -> ]:
     morf[target, transety -> ];
     comorf[target, transety -> ];
     unit[t, transety -> ], <<, mode[ -> t], >>;

     coll, collateral list[1, target, transety -> ],
       check[target is void, structured, or rows of mode],
       collend;
     vacuum, /mode[target], check[target begins with "row"];
     check[mode = "void"],
       from, unety["integral", "forbid" -> ],
       by, unety["integral", "forbid" -> ],
       to, unety["integral", "forbid" -> ],
       sr, for, defining identifier,
       sr, while, unety["boolean", "forbid" -> ],
       do, sr, unit["void", "forbid" -> ],
       er, er, er, od;

     series[target, transety -> ];

     /":=", unit, unit, <<, ":=", >>;
     /is, unit[M, "forbid" -> ], unit[M, "forbid" -> ], mode[ -> M], <<,
         is, >>;
     /isnt, unit[M, "forbid" -> ], unit[M, "forbid" -> ], mode[ -> M],
         <<, isnt, >>;
     text, mode, unit, endtext;
     textp, mode, "(", sr, formal parameters, ")", unit, er, endtextp;
     /jump, /mode[target], identifier, <<, goto, >>;
     skip, /mode[target -> ];
     nihil, /mode[target -> ], check['target' begins with "reference
         to"];
     /dyad, /definition number[r -> ], unit[M1, "forbid" -> ], operator,
          unit[M2, "forbid" -> ], <<, definition number[ -> n], mode[ ->
          M1], mode[ -> M2], d yad, >>;
      /monad, /definition number[n -> ], operator, unit[M, "forbid" -> ],
          <<, definition number[ -> n], mode[ -> M], monad, >>;
      /gen, leap, declarer, <<, gen, >>;
      /of, tag, unit[M, transety -> ],
        check[if transety = "forbid" then M does not begin with
          "reference to flexible" else ok fi], mode[ -> M], <<, of, >>;
      /slice, unit[M, transety -> ],
        check[if transety = "forbid" then M does not begin with
          "reference to flexible" else ok fi],
        mode[ -> M], number, "[", trimscript list, "]", <<, slice, >>;
      /call, unit, number, "(", unit list, ")", <<, call, >>;
      unit, <<, cast, >>;
      denoter;
      "$", collection list, "$";
      identifier.

conformer list[m, t -> ]:
      conformer[m, t -> ];
      conformer[m, t -> ], ",", conformer list[m, t -> ].
conformer[m, t -> ]:
      confrange, sr, declarer, identifier, unit[m, t -> ], er,
          endconfrange;
      confnorange, declarer, unit[m, t -> ], endconfnorange.

unit list[mode, t -> ]:  unit[mode, t -> ];
      unit[mode, t -> ], ",", unit list[mode, t -> ].

collateral list[i, Target, transety -> ]:
      unit[submode(target, i), transety];
      collateral list[i+1, target, vansety -> ], ",",
          unit[submode(target, i), transety -> ].
      [ ???  check transience for correctness ???  ]
[Explanation of "submode":
      if target is some "ROW MODE", then submode(target, i) = "mode".
      if target is some STRUCTured, then submode(target, i) is the i-th
          last field mode.
      if target is "void", then submode(target, i) is "void".
      otherwise, produce an error message and yield "erroneousmode".
      ]

morf[target, transety -> ]:
      /insert coercions[coercions -> ], unit[apriori, transety2 -> ],
          coerce[apriori, "morf", target, transety -> transety2,
          coercions], <<, mode[ -> apriori], morf, >>.

comorf[target, transety -> ]:
      /insert coercions[coercions -> ], unit[apriori, transety2 -> ],
          coerce[apriori, "comorf", target, transety -> transety2,
          coercions], <<, mode[ -> apriori], comorf, >>.

identifier:
      /tag, tag, definition number, mode, <<, tag, >>.

formal parameters:
     defining identifier, mode;
     defining identifier, mode, ",", formal parameters.

trimscript list:
     trimscript;
     trimscript, ",", trimscript list.
trimscript:
     sub, unit;
     trim, unety, ":", unety;
     trim, unety, ":", unety, "@", unit;
     trim, missing, "@", unit.

series[target, transety -> ]:
     series, preludety[target, transety -> ],
     unit[target, transety -> ], endseries.
preludety[target, transety -> ]:  ;
     preludety[target, transety -> ], unit["void", "forbid" -> ], ";";
     preludety[target, transety], unit[target, transety -> ], exit;
     preludety[target, transety], ":", defining identifier, ":";
     preludety[target, transety], declaration, ";".

declaration:  single declaration;
     single declaration, ",", declaration.
single declaration:
     iddec, <<, declarer, >>, indicator, "=", unit, endiddec;
     modec, mode indication, "=", declarer, endmodec;
     vardec, declarer, vardefs, endvardec;
     priodec.
vardefs:  vardef;
     vardef, ",", vardefs.
vardef:
     identifier, noinit;
     identifier, ":=", unit.

declarer:  /acdecer, mode, <<, acdecer, >>;
     /videcer, mode, <<, videcer, >>;
     /fodecer, mode, <<, fodecer, >>;
     /modind, /mode[m -> ], mode indication, <<, mode[ -> m], modind,
         >>;
     /struct, /mode[m -> ], "(", fields, ")", <<, mode[ -> m], struct,
         >>;
     /array, /mode[m -> ], "[", actual bounds, "]", declarer, <<, mode[
         -> m], array, >>;
     /flex, /mode[m -> ], declarer, <<, mode[ -> m], flex, >>.
actual bounds:
     actual bound pair;
     actual bound pair, ",", actual bounds.
actual bound pair:
     unit;
     unit, ":", unit.

indicator:
     indication;
     identifier.

check[X]:
```

[Perform the check X, producing an error message if the check
    fails.].