```
=======================================
SYNTAX ERROR RECOVERY

Not ok to announce a single syntax error and stop parsing.
Have to recover and continue, to find additional errors.

"Phrase-level" recovery defines a _set_ of well-defined places to back
out to: e.g. end of current expression, statement, or declaration.

Wirth's formalization for recursive descent
  - On a token mismatch, insert what you expect and print an error message
  - On a null prediction in the RD routine for nonterminal A
    (no matching label in switch)
    delete tokens until you see something in FIRST(A) or FOLLOW(A)
          (also stop if you see $$)
      if the FIRST case, restart the current routine
        - assume what we saw was garbage and can be ignored
      if the FOLLOW case, return
        - assume what we saw was the desired nonterminal, garbled

So the RD routine for statements might be

    procedure stmt
        if not (input_token ∈ FIRST(stmt))  // NB: stmt cannot derive ε
            report_error()
            repeat
                get_next_token()
            until input_token ∈ (FIRST(stmt) U FOLLOW(stmt) U {$$})
        case input_token of
            id    : match(id); match(:=); expr()
            read  : match(read); match(id)
            write : match(write); expr()
            // no else clause needed

That initial if clause can of course be abstracted out into a routine
that is then called at the top of each RD routine:

    procedure check_for_error(sym)
        if not (input_token ∈ FIRST(sym) or sym =>+ ε)
            report_error()
            repeat
                get_next_token()
            until input_token ∈ (FIRST(sym) U FOLLOW(sym) U {$$})

Simpler strategies are possible.
Here's one based on exceptions that avoids need for error handling logic
in all RD routines:
  - On a token mismatch we still insert what we expect
    (and print an error message)
  - On a null prediction, we throw a syntax_error exception.
  - Exceptions are caught by handlers in some subset of RD routines --
    "phrases" in the grammar -- statement, declaration, block, function, etc.

E.g.: replace

    procedure stmt
        case input_token of
            id    : match(id); match(:=); expr()
            read  : match(read); match(id)
            write : match(write); expr()
            else    error()
with

    procedure stmt
        try
            case input_token of
                id    : match(id); match(:=); expr()
                read  : match(read); match(id)
                write : match(write); expr()
                else    throw syntax_error
        except when syntax_error =>
            loop
                if input_token in FIRST(stmt)
                    stmt()          -- try again
                    return
                elsif input_token in (FOLLOW(stmt) U {$$})
                    return          -- caller can probably make progress
                else get_next_token()
                    -- NB: get_next_token is normally called only in match()

NB: accepting a token in FIRST(stmt) and restarting may or may not
be a good idea.  It's always a good idea in Wirth's algorithm, because
we detect errors only at the beginning of the RD routine.  But with
exceptions we may land in the handler halfway through the construct (in
this case, stmt).  At that point we may have already accepted a big
chunk of the statement.  Starting over implicitly means silently
ignoring what we've seen of the statement so far.  It may be better just
to delete to what we hope is the end -- that is, to write the simpler

    procedure stmt
        try
            ...                 -- code to parse a statement
        except when syntax_error =>
            while input_token not in (FOLLOW(stmt) U {$$})
                get_next_token()

Fancier strategies are also possible.  Fischer, Milton, and Quiring
developed a particularly pretty "tunable", locally-least-cost recovery
mechanism for table-driven LL(1) (See the book).




-----------------------------------------
The immediate error detection problem
and context-sensitive follow sets

Several error-recovery mechanisms, including the version of Wirth's
described above, will sometimes predict an epsilon production when
calling routines are doomed to discover an error.
    Arguably, we should detect the error before generating epsilon.
    That way we have more context with which to craft recovery.

Example from the book, in the calculator language:

Y := (A * X X*X) + (B * X*X) + (C * X)
           ^ There's a problem here (missing '*' in polynomial).
             Can we tell?

When we're at the point shown in the parse, what recursive descent
routines are active?

                    (dot shows where we are inside)
    program                 P -> . SL $$
    stmt_list               SL -> . S SL
    stmt                    S -> id := . E
    expr                    E -> . T TT
    term                    T -> . F FT
    factor                  F -> ( . E )
    expr                    E -> . T TT
    term                    T -> F . FT
    factor_tail             FT -> * F . FT
    factor_tail             FT -> ?

Now ID can follow expr in some programs (e.g. A := B C := D), and an
expr can end with a factor_tail, so ID is in FOLLOW of factor_tail.  And
since factor_tail and term_tail can generate epsilon, the "obvious"
thing is to return from FT twice, return from T (which thinks it's
done); call from E to TT; return from TT; and return to F _all_ without
detecting an error of any kind_.  At this point we'll (finally) get a
mismatch between ID and ).  Unfortunately we won't have much information
to work with at that point, and won't be able to make as good a
recovery as we would have liked.

Specifically: match will insert a right paren, allowing F to complete
and return.  T will call FT, which will see X on the input, which is in
FOLLOW(FT), so it will predict and epsilon production and return,
allowing T to return.  E will likewise call TT, which predicts epsilon
and returns, allowing E to return, at which point S will complete and
return, allowing SL to make a recursive call.  Now we have

    X*X) + (B * X*X) + (C * X)

on the input, but we've left the context in which we could continue to
parse more pieces of an expression.

SL will predict S -> id := E.  We'll match id (X), insert :=, call E,
T, and then F.  F will predict F -> id, match X, then return
all the way back to

    SL -> S . SL

at which point we'll make another recursive call to SL and run into
trouble with ) on the input.  We'll delete the ), predict S -> id := E,
and soon run into trouble again when we see * instead of := on the
input.  When the dust settles, our final "correction" will be

    Y := (A * X)  X := X  B := X * X  C := X

If we were smarter, when FT saw X way back at the beginning it would
know that an ID can't follow a factor_tail _in this particular context_
(where we're inside a parenthesized expression, not at the end of an
assignment).  Good error recovery algorithms take this into account.
Wirth showed how to do it in the (better version of) his error-recovery
algorithm for recursive descent.  He adds a _context-sensitive follow set_
parameter to every R.D. subroutine, and uses these, rather than global
FOLLOW, to predict epsilon productions.

So, for example, when F calls E in the example above, it would pass as
E's follow set only { ')' }.  When E calls T it would pass that same
set, plus FIRST(TT) -- i.e., { ')', '+', '-' }.  When T calls FT it
would pass what it, itself, was given, namely { ')', '+', '-' }.
When FT calls itself recursively it would pass this same set yet again.
When the nested FT sees 'id' on the input, it would know there was a
problem.  It would delete the id.  The subsequent * is in FIRST(FT), so
all would be well at that point.  Recognizing the problem early allows
the parser to, effectively, "correct" the input into

    Y := (A * X*X) + (B * X*X) + (C * X)

Not "right", but certainly better.

Generalizing, our top-of-routine error checker now looks like this:

    procedure check_for_error(sym, LAset)
        if not (input_token ∈ FIRST(sym)
                or (sym =>+ ε and input_token ∈ FOLLOW(sym))
            report_error()
            repeat
                get_next_token()
            until input_token ∈ (FIRST(sym) U FOLLOW(sym) U {$$})


We can do something very similar with exception-based recovery, if we
pass context-sensitive FOLLOW sets into appropriate RD routines.

One can also do something similar in table-driven parsers, but for
these there's an even easier alternative: go ahead and do the
epsilon productions, but remember one did so, and when a problem
arises, restore the stack to where it was when the error _should_
have been noticed, and recover from there instead.  There isn't a
good analogue of this approach for the recursive descent case: we
can't "undo" having returned from a bunch of R.D. routines the way
we can restore the explicit stack of the T.D. parser.

ANTLR, by default, uses global FOLLOW sets and Java/C++/C# exception
handlers, but the compiler writer can (by hand) write smarter handlers.

FMQ (a parser generator developed at the Univ. of Wisc., which we used
many years ago) buffered epsilon productions and then undid them,
putting context back on the stack.  FMQ also implements tunable "locally
least cost" repair.
```