==
TABLE-DRIVEN LL PARSING

Table-driven LL parsing is essentially a different way to think about recursive descent. You have a big loop in which you repeatedly look up an action in a two-dimensional table based on current leftmost non-terminal and current input token. The actions are (1) match a terminal, (2) predict a production, or (3) announce a syntax error.

  - When you predict a production, you replace its LHS (currently at top of stack) with the symbols of the RHS, so the new TOS is the first symbol of the RHS.

  - This means the stack always contains what you expect to see in the future.

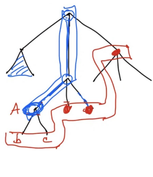
```
                    grammar:
                        program       ->  stmt_list $$
                        stmt_list     ->  stmt stmt_list | ε
                        stmt          ->  ID := expr | READ ID | WRITE expr
                        expr          ->  term term_tail
program:                term_tail     ->  add_op term term_tail | ε
    read A              term          ->  factor fact_tail
    read B              fact_tail     ->  mult_op factor fact_tail | ε
    sum := A + B        factor        ->  ( expr ) | ID | LITERAL
    write sum           add_op        ->  + | -
    write sum / 2       mult_op       ->  * | /

    stack                                       remaining input
    -----                                       ---------------
    pgm                                         read A read B sum ...
    stmt_list $$                                read A read B sum ...
    stmt stmt_list $$                           read A read B sum ...
    READ ID stmt_list $$                        A read B sum := A ...
    ID stmt_list $$                             read B sum := A + ...
    stmt_list $$                                read B sum := A + ...
    stmt stmt_list $$                           read B sum := A + ...
    READ ID stmt_list $$                        B sum := A + B ...
    ID stmt_list $$                             sum := A + B write ...
    stmt_list $$                                sum := A + B write ...
    stmt stmt_list $$                           sum := A + B write ...
    ID := expr stmt_list $$                      := A + B write sum ...
    := expr stmt_list $$                        A + B write sum ...
    expr stmt_list $$                           A + B write sum ...
    term term_tail stmt_list $$                 A + B write sum ...
    factor fact_tail term_tail stmt_list $$     A + B write sum ...
    ID fact_tail term_tail stmt_list $$         + B write sum / 2 $$
    fact_tail term_tail stmt_list $$            + B write sum / 2 $$
    term_tail stmt_list $$                      + B write sum / 2 $$
    add_op term term_tail stmt_list $$          + B write sum / 2 $$
    + term term_tail stmt_list $$               B write sum / 2 $$
    term term_tail stmt_list $$                 B write sum / 2 $$
    factor fact_tail term_tail stmt_list $$     B write sum / 2 $$
    ID fact_tail term_tail stmt_list $$         write sum / 2 $$
    fact_tail term_tail stmt_list $$            write sum / 2 $$
    term_tail stmt_list $$                      write sum / 2 $$
    stmt_list $$                                write sum / 2 $$
    stmt stmt_list $$                           write sum / 2 $$
    WRITE expr stmt_list $$                     sum / 2 $$

    ...     etc

    stmt_list $$                                $$
    $$
```

Remember: the stack contains all the stuff you expect to see between now and the end of the program -- what you ***predict*** you will see.
These correspond in a recursive descent parser to the concatenation of the remainders of the current case arm in all the RD routines on the current call chain.



LL PARSER GENERATORS

The algorithm to build PREDICT sets is tedious (for a "real" sized grammar), but relatively simple.

    (1) compute FIRST sets and EPS values for symbols
    (2) compute FOLLOW sets for non-terminals (separate from epsilon)
        (this requires computing FIRST sets for some ***strings***)
    (3) compute PREDICT sets for productions
        (this requires computing EPS for some ***strings***)

where

    EPS(α) == if α =>* ε then true else false
    FIRST(α) == {c : α =>* c β}
    FOLLOW(A) == {c : S =>* α A c β}
    PREDICT(A -> α) == FIRST(α) U (if EPS(α) then FOLLOW(A) ELSE ø)

Steps (1), (2), and (3) begin with "obvious" facts, and use them to deduce more facts, until nothing new is learned in a full pass through the grammar.

What is obvious? At *a* minimum:
    If A -> ε, then EPS(A) = true
    c in FIRST(c)

How to deduce?
    If EPS(α) = true and A -> α, then EPS(A) = true
    If A -> B β, then FIRST(A) ⊃ FIRST(B)
    If A -> α B β, then FOLLOW(B) ⊃ FIRST(β)
    If A -> α B (or A -> α B β and EPS(β) = true)
        then FOLLOW(B) ⊃ FOLLOW(A)

This last one is tricky. It's ***not*** true the other way around.
That is, A -> α B does ***not*** imply that FOLLOW(A) ⊃ FOLLOW(B).
Consider our calculator grammar.
    ')' is in FOLLOW(E), because F -> (E)
    $$ is in FOLLOW(S), because P -> SL $$, SL -> S SL, and SL -> ε

Now consider the production S -> write E.
The fact that $$ is in FOLLOW(S) means than $$ is in FOLLOW(E).
But the fact that ')' is in FOLLOW(E) does ***not*** mean that
')' is in FOLLOW(S).

Put another way, ')' is in FOLLOW(E) in the context where E was generated from F, but ***not*** necessarily in the context where E was generated from S.

If any token belongs to the PREDICT set of more than one production with the same lhs, then the grammar is not LL(1).
A conflict can arise because
    some token c can begin more than one rhs, or
    c can begin one rhs and can also appear ***after*** the LHS in some
        valid program, and one possible RHS is epsilon.

Examples 2.33-2.35 in the book work through the generation of a table-driven parser for the calculator language.

    Fig. 2.22 shows the "obvious" facts in the calculator grammar
    Fig. 2.23 shows the generated FIRST, FOLLOW, and PREDICT sets
    Fig. 2.20 contains the resulting parse table
    Fig. 2.19 contains a parser driver that reads the parse table

Again, the algorithm to generate the parse table

    (1) computes FIRST sets and EPS values for symbols
    (2) computes FOLLOW sets for non-terminals (separate from epsilon)
        (this requires computing FIRST sets for some ***strings***)
    (3) computes PREDICT sets for productions
        (this requires computing EPS for some ***strings***)

Here are the details:

```
    -- EPS values and FIRST sets for all symbols:
        for all terminals c
            EPS(c) := false;  FIRST(c) := {c}
        for all non-terminals X
            EPS(X) := if X -> ε then true else false
            FIRST(X) := ø
        repeat
            <outer> for all productions X -> Y1 Y2 ... Yk
                <inner> for i in 1..k
                    add FIRST(Yi) to FIRST(X)
                    if not EPS(Yi) (yet) then continue outer loop
                EPS(X) := true
        until no further progress

    -- Subroutines for strings, similar to the inner loop above:
        function string_EPS(X1 X2 ... Xn):
            for i in 1..n
                if not EPS(Xi) then return false
            return true

        function string_FIRST(X1 X2 ... Xn):
            return_value := ø
            for i in 1..n
                add FIRST(Xi) to return_value
                if not EPS(Xi) then return

    -- FOLLOW sets for all symbols:
        for all symbols X, FOLLOW(X) := ø
        repeat
            for all productions A -> α B β
                add FIRST(β) to FOLLOW(B)
            for all productions A -> α B
                or A -> α B β, where string_EPS(β) = true
                add FOLLOW(A) to FOLLOW(B)
        until no further progress

    -- PREDICT sets for all productions:
        for all productions A -> α
            PREDICT(A -> α) := string_FIRST(α)
                U (if string_EPS(α) then FOLLOW(A) else ø)
```

At the end, the grammar is LL(1) iff all the PREDICT sets for productions with the same LHS are disjoint

----
SYNTAX ERROR RECOVERY (reprise)

Natural adaptation of phrase-level recovery to table-driven top-down parsing:
  - When we encounter an error in match (TOS is a token that doesn't match the input), we print a message and pop the stack (pretend to have seen the desired token).

  - When we encounter an error entry in the table (non-terminal A at TOS), we delete tokens until we find something in FIRST(A) or FOLLOW(A).
    If in FIRST(A), we continue the main loop of the driver. If in FOLLOW(A), we pop the stack first.

    ($$ is a special case: if we see that, we pop the stack and continue the main loop.)

    More generally, we may define a set of "starter symbols" that are too dangerous to delete (begin, left paren, procedure, ...), because they are likely to presage subsequent structure. Treat them like $$.
    Hopefully they'll be in FIRST of something deeper in the stack. If not, we'll eventually end up with $$ on the stack and remaining input, at which point we print a message and die.

As in the recursive descent case, we probably want to consider the immediate error detection problem.

Adding context-sensitive follow sets to the stack is a nuisance, however.

Much easier, when we predict an epsilon production, to remember that we did so, and buffer what we popped off the stack.
  - If we accept a new token of real input, we can toss the buffer.
  - If we run into an error before then, we put the buffered symbols back on the stack and initiate error recovery as shown above.