```
=====================================
LL PARSING and RECURSIVE DESCENT

We can implement top-down parsing in two ways:
  - recursive descent parser
      - written by hand or automatically
  - parse table and a driver
      - written automatically

We'll consider the table-driven option more in a bit.
If you took 173 you probably saw recursive descent; this is a review.

Key idea: set of mutually recursive subroutine, one for each nonterminal.
Each such routine is responsible for discovering a subtree of the parse
tree, rooted at the symbol for which it is named.

Also need a _match_ routine:
    takes a token name as argument and reads a matching token from the
    input stream, or announces an error if it can't.

(How to handle errors comes in the next lecture.  For now, let's assume
we just quit.)

Consider recursive descent routines for the calculator language:

The parser begins by calling the following subroutine:

    procedure pgm
        case input_token of
            id, read, write, $$ : stmt_list; match($$)
            else                  error

Other subroutine include:

    procedure stmt_list
        case input_token of
            id, read, write : stmt; stmt_list
            $$              : skip // epsilon
            else              error

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

    procedure expr
        case input_token of
            id, literal, ( : term; term_tail
            else             error

    procedure term
        case input_token of
            id, literal, ( : factor; fact_tail
            else             error

    procedure term_tail
        case input_token of
            +, -                    : add_op; term; term_tail
            ), id, read, write, $$ : skip // epsilon
            else                     error

    etc.

Each routine knows that it's expecting to see
  - the _yield_ of the symbol for which it is named
It needs to
(1) choose a production with which to generate the symbol's children in
    the parse tree
      - makes this choice based on the upcoming token from the scanner
(2) parse those children one by one
      - match any that are terminals
      - call the appropriate RD routine to parse any that are nonterminals

So how exactly do we know (in a complicated grammar) which production to
use, given an expected nonterminal (root of to-be-fleshed-out subtree)
and upcoming token?
That is, how to label the arms of the switch statements?

-----------------------------------------------
PREDICT Sets

If a RHS can start with a given token (directly or indirectly), the
appearance of that token *predicts* its rhs.

If the rhs is epsilon (or something that can derive epsilon), any token
that can follow the LHS anywhere in the grammar predicts the epsilon
production.

An LL(1) parser generator constructs these "predict sets" for you.
We'll consider the algorithm in a future lecture.  It depends on the
following definitions:

    FIRST(α) ≡ {c : α =>* c β}
    FOLLOW(A) ≡ {c : S =>+ α A c β}
    PREDICT(A -> α) ≡ FIRST(α)
        U (if α =>* ε then FOLLOW(A) else ∅)

Here -> is the familiar "goes to" symbol used in productions.
=> means "derives" - can be replaced by.  Note that its LHS doesn't have
    to be a single symbol.
=>* means "derives in zero or more steps"
=>+ means "derives in one or more steps"

FIRST sets capture the "RHS can start with 'c'" case.
FOLLOW sets capture the "RHS can generate ε" case.

NB: conventional notation uses
    lower case letters near the beginning of the alphabet for terminals
    lower case letters near the end of the alphabet for _strings_ of terminals
    upper case letters near the beginning of the alphabet for non-terminals
    upper case letters near the end of the alphabet for arbitrary symbols
    greek letters for arbitrary _strings_ of symbols

*** In a recursive descent parser, if c ∈ PREDICT(A -> α), then the RD
    routine for A will predict A -> α when it sees c on the input.

The calculator language is simple enough that one can figure these out
more or less by inspection.  "Real" languages are too complex for that
to be a reasonable task.  We need an algorithm (stay tuned).

----------------------------------------
MAKING A GRAMMAR LL

Note the implicit assumption that the choice among productions
A -> α and A -> β is always uniquely determined.
What if there is more than one production w/ a LHS of A and a RHSs that
can start w/ the same nonterminal?
Or two RHSs than can generate epsilon?
Or a RHS that can start with c _and_ a RHS that generate epsilon, when c
is in FOLLOW(A)?

In this case the grammar is not LL(1) - by definition.

If you're trying to write an LL(1) grammar, you probably want to avoid
_left recursion_ and _common prefixes_

left recursion
    example
        id_list => ID | id_list , ID
    convert this to
        id_list => ID id_list_tail
        id_list_tail => , ID id_list_tail | ε

common prefixes
    example
        stmt => ID := expr | ID ( arg_list )
    convert this to
        stmt => ID id_stmt_tail
        id_stmt_tail => := expr | ( arg_list )

Both left recursion and common prefixes can be removed mechanically.
Note, however, that there are infinitely many non-LL _languages_, and the
mechanical transformations work on them just fine, so removing these is
necessary but not sufficient to make a grammar LL(1).
Fortunately, the cases that arise in practice, however, can generally be
handled with kludges.

A famous example was the if-then-else statements of Algol-60 and Pascal.
Does

    if A < B then if C < D then X := 1 else X := 2

mean

    if A < B then
        if C < D then
            X := 1
        else
            X := 2

or

    if A < B then
        if C < D then
            X := 1
    else
        X := 2

The hack for top-down parsers was to force the first interpretation with
a bit of special-case code.  If the programmer wanted the sescond
interpretation they needed to type

    if A < B then begin
        if C < D then
            X := 1
    end
    else
        X := 2

Languages since 1970 have fixed this with 'elsif' and 'endif'/'fi'.

    if A < B then
        if C < D then
            X := 1
        fi
    else
        X := 2
    fi
```