==========
TOP-DOWN AND BOTTOM-UP PARSING

*** A LL family parser builds a leftmost derivation from the top down.
*** A LR family parser builds a rightmost derivation from the bottom up.

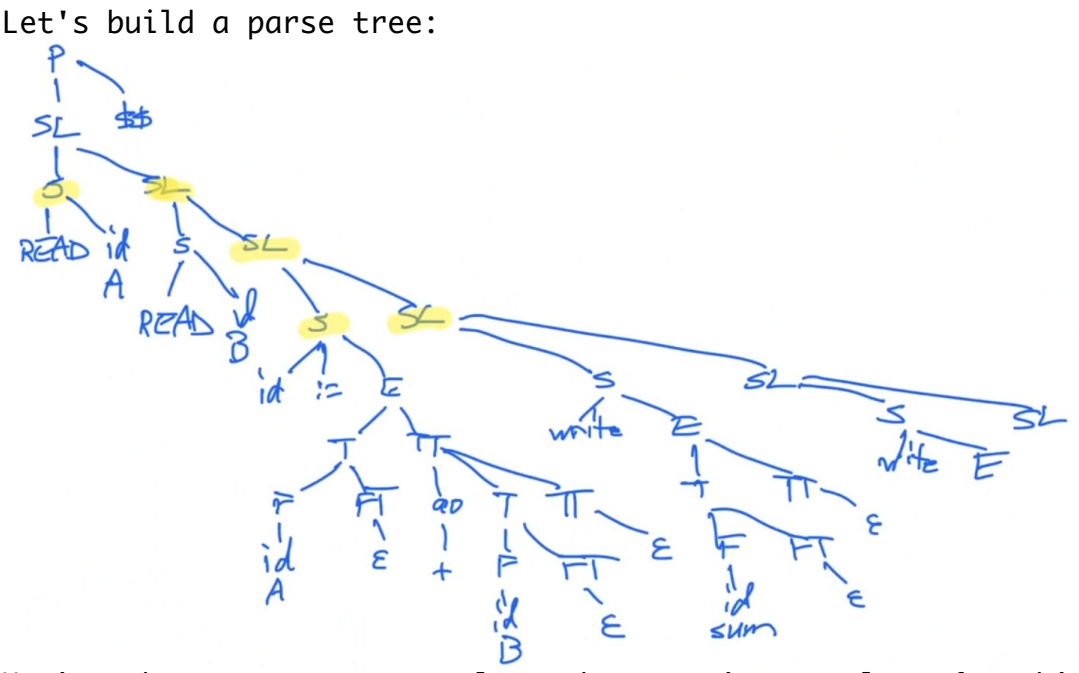
How do we parse a string with the top-down grammar? You can get the general idea by building the parse tree incrementally by hand:

> Start at the top and *predict* needed productions on the basis of the current left-most non-terminal in the tree and the current input token.

Consider our example program again, together w/ the top-down grammar:

```
                                    P  ->  SL $$
                                    SL ->  S SL | ε
                                    S  ->  ID := E | READ id | WRITE E
                                    E  ->  T TT
                                    TT ->  ao T TT | ε
    read A                          T  ->  F FT
    read B                          FT ->  mo F FT | ε
    sum := A + B                    F  ->  ( E ) | id | lit
    write sum                       ao ->  + | -
    write sum / 2   $$              mo ->  * | /
```

Let's build a parse tree:



Notice that at every step along the way, it was clear (unambiguous) what to do.

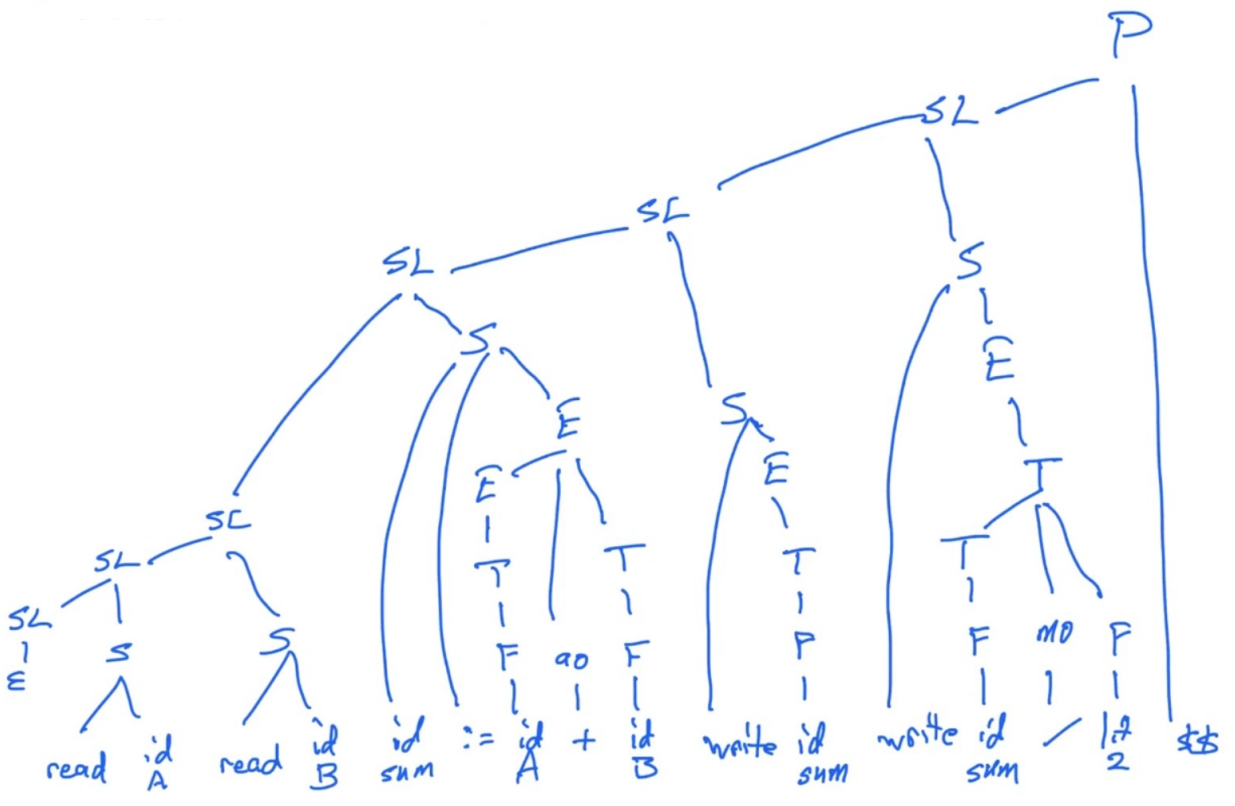
We can also get a sense of the bottom-up case with an example, but it won't be as obvious what's going on.

> Just as a scanner is based on a finite automaton,
> a parser is based on a _pushdown_ automaton
> - basically a finte automaton with a stack
> - makes a decision based on input, state, and top-of-stack symbol
> - chooses a new state and may push or pop the stack
>
> A top-down parser has a trivial state machine.
> - makes all decisions based on input and top-of-stack symbol until it sees end-of-file, at which point it switches to a final state if the stack looks right (more on this later)
>
> A bottom-up parser has a complex state machine.
> - uses current state to make decisions
> - I won't be showing you the state machine in this example

Consider our example program again, together w/ the bottom-up grammar:

```
                                    P  ->  SL $$
                                    SL ->  SL S | ε
                                    S  ->  ID := E | READ id | WRITE E
    read A                          E  ->  T | E ao T
    read B                          T  ->  F | T mo F
    sum := A + B                    F  ->  ( E ) | id | lit
    write sum                       ao ->  + | -
    write sum / 2   $$              mo -> * | /
```

Let's build a parse tree:



The power of bottom-up parsing comes from its ability to recognize things "after the fact," rather than predicting them up front. This same power explains why error messages and special-case hacks are harder to implement in the bottom-up-case: the parser isn't always sure what's going on until after it's finished.