### 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:

```
grammar:
  T -> T . mul_op F       OW reduce (by 3, recognizing E)
  E -> E add_op T .       on mul_op shift and goto 11
  F -> ( E . )            on ) shift and reduce (by 3, recognizing F)
  F -> . LITERAL          on LITERAL shift and reduce (by 1, recognizing F)
  S -> . WRITE E          on WRITE shift and goto 4
```

```
0 stmt_list 1 WRITE 4 expr 6                  $$
0 stmt_list 1 WRITE 4                         T / 2 ... sum := A + B
0 stmt_list 1 ID 2                            := A + ...
```

illustrate it with growing-together pieces of the tree >> production to simplify the presentation.

factor        ->  ( expr ) | ID | LITERAL

state diagram looks just like a DFA state diagram, except the arcs are repeatedly inspects a two-dimensional table to find out what action to take.

```
loop
2   stmt_list     ->  stmt stmt_list |
```

```
4-state minimal DFA
```

```
FIRST(FT) or FOLLOW(FT) \{id} = {+, -, ), read, write, $$}. We can do much better with _context-sensitive follow sets_, as in Wirth's formalization for recursive descent
```

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

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

```
- The parse table predicts FT -> stack:  FT TT ) FT TT SL $$     input:  X * X) + (B * X * X) + (C * X) $$
```

```
^ There's a problem here (missing '*' in polynomial).
```

If A -> TT ->  ao T TT |  id_list => ID | id_list , ID

```
this is a problem
```

```
We detect errors only at the beginning of the RD routine.  But with NB: accepting a token in FIRST(stmt) and restarting may or may not
```

```
<outer> for all productions X -> Y1 Y2 ... Yk
```

```
FIRST(X) :=
```

```
// no else clause needed
```

```
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 $$
SL -> S SL | e
S -> ID := E | READ id | WRITE E
E -> T TT
TT -> ao T TT | e
read A
T -> F FT
read B
FT -> no F FT | e
sum := A + B
F -> ( E ) | id | lit
write sum
aa -> + 1
write sum / 2 $$
no -> * 1 /
```

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 finite automaton with a stack - makes a decision based on input, stack, 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 $$
SL -> S SL | e
S -> ID := E | READ id | WRITE E
E -> T TT
TT -> ao T TT | e
read A
T -> F FT
read B
FT -> no F FT | e
sum := A + B
F -> ( E ) | id | lit
write sum
aa -> + 1
write sum / 2 $$
no -> * 1 /
```

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.