Notes for CSC 2/454, Sept. 11, 13, and 18, 2023

==========
SCANNING

Scanner is responsible for
  tokenizing source
  removing comments
  saving text of identifiers, numbers, strings
  saving source locations (file, line, column) for error messages

  a DFA for identifiers:

Can be built by hand (ad hoc) or automatically from regular expressions (REs)
  ad-hoc generally yields the fastest, most compact code
    by doing lots of special-purpose things
  automatically-generated scanners can come close, though
    and are easy to develop and change

A scanner generator builds a DFA automatically from a set of REs
Specifically, it constructs a machine that accepts the “language”

  identifier | int const | real const | comment | symbol | ...

In other words, a real scanner accepts the alternation of a language's
tokens, with a separate final state for each.
We run the scanner over and over to get one token after another.

(Note that theoreticians use “language” to mean a set of strings --
not nec. the valid programs of a programming language.)

Nearly universal rule:
always take the longest possible token from the input
thus `foobar` is `foobar` and never `f` or foo or `foob`
more to the point, 3.14159 is a real const and never 3, ., and 14159

An RE generates a regular language; a DFA *recognizes* it.

The standard Unix lex (flex) outputs C code.
Some other tools produce numeric tables that are read by a separate driver.
The table is the transition function
two-dimensional array indexed by current state and input character
entries specify
next state
whether to keep scanning, return a token, or announce an error

Longest-possible token rule means we return only when the next character can't be used to continue the current token.
That next character must generally be saved for the next token.

In some cases you may need to peek at more than one character of lookahead in order to know whether to proceed.

In Ruby, when you have a 3 and you see a dot, do you proceed (in hopes of getting 3.14) or do you stop (in fear of getting 3..5 or 3...5)?

In messier cases, you may not be able to get by with any fixed amount of lookahead. In Fortran IV (c. 1962), for example, one had

```
DO 5 I = 1,25    loop
DO 5 I = 1.25    assignment
DO 5,I = 1,25    alternate syntax for loop, f77
```

For most languages it suffices to remember we were in a potentially final state, and save enough information that we can back up to it if we get stuck later.
For some languages (famously, Fortran), that isn't enough.
Sometimes need semantic information in order to scan (yuck).

Building a scanner from regular expressions

multi-step process

1) write REs by hand, including for whitespace and comments, but with identifiers and reserve words (keywords) combined
2) build NFA from REs
3) build DFA from NFA
4) minimize DFA
5) add extra logic to
    - implement the longest-possible-token rule, with backup
    - discard white space and comments (i.e., start over when you realize that's what you found)
    - distinguish reserve words from identifiers
    - save text of “interesting” tokens
    - tag returned tokens with location and text
    - return an extra $$ token at end-of-file

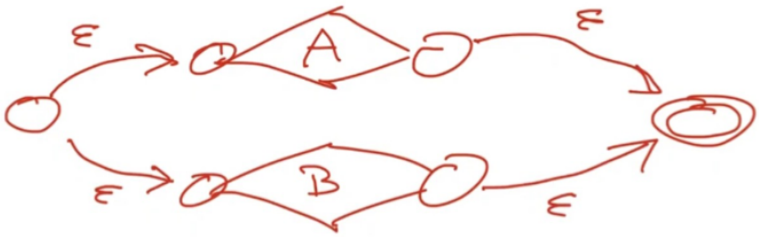
In most compilers, the parser drives the front end.
The scanner is a subroutine (function) called by the parser.

Step (2) above is inductive. It starts with a trivial DFA to accept a single character:

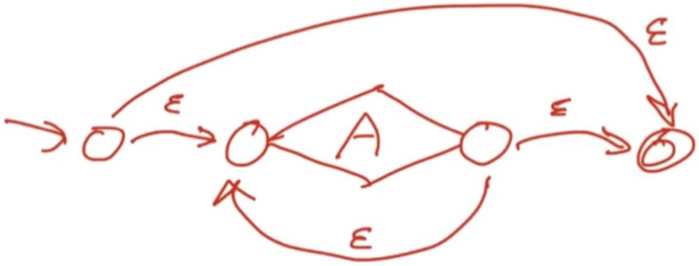
Note that this machine has a single start state, a single final state, no transitions into the start state, and no transitions out of the final state. We'll maintain these invariants in three inductive steps:

Concatenation (A B):

Alternation (A | B):



Kleene closure (A*):



Step (3) uses what's called a “set of subsets” construction.
Step (4) divides the states of an initial DFA into progressively finer equivalences classes, until it can prove that additional refinement makes no difference.

Example 1 (in the book): real numbers (no exponential notation)

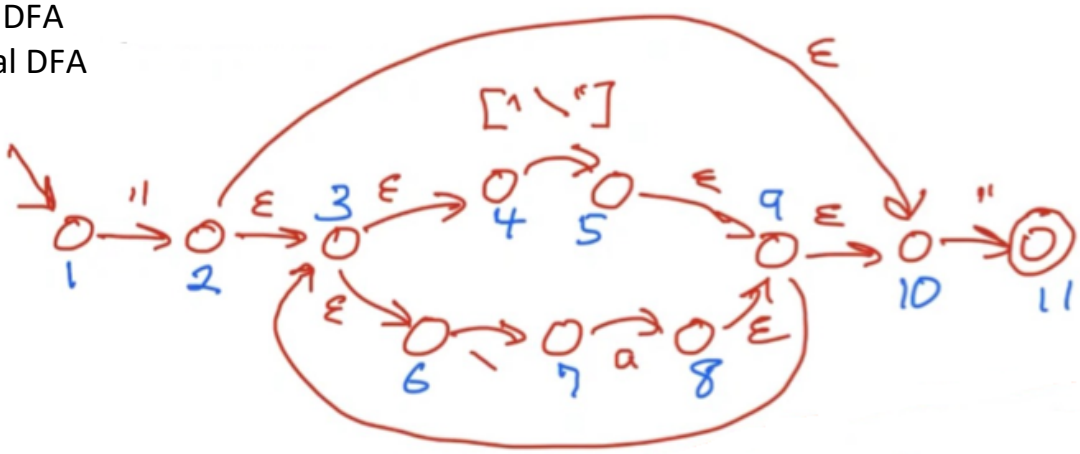
RN = d*(.d|d.)d*

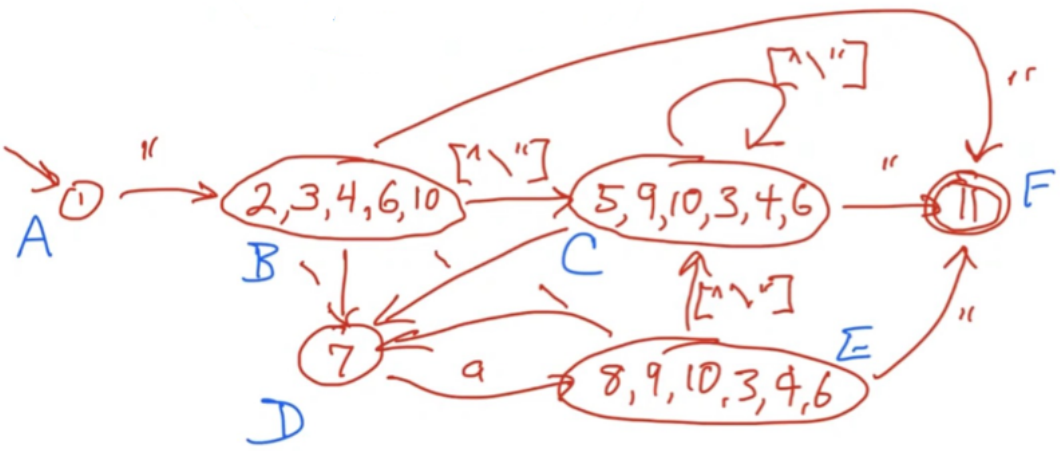
14-state NFA results from construction
5-state subset DFA
4-state minimal DFA

Example 2: character strings with optional backslash-escaped quotes

S = “ ([^\”] | \a)* “ “a” for anything

11-state NFA results from construction
6-state subset DFA
4-state minimal DFA





A, B, C, D, E		F	final
AD	BCE	F	on "
A	D	BCE	F	on \

NB:

(1) There's nothing magic about the characters we choose to consider as possible reasons to split a group. In a mechanical tool, we'd simply iterate over all characters in the alphabet. Many wouldn't teach us anything.

(2) Real scanner accepts alternation of tokens, with a separate final state for each. Scanner generator

- starts with NFAs for all the separate tokens
- creates a new start state with an E-transition to the start state of each token NFA

- turns that into a DFA
- runs the minimization algorithm starting not with two classes (final and non-final) but with K+1: non-final, final for token T1, final for token T2, ..., final for token TK.

Suggestion: try Exercise 2.5 in the book: extend Examples 1.13-15 to build a minimal DFA for intergers *and* decimals together.

In our two examples, the DFA was smaller than the original NFA.
Is that always the case?

No! Quite the contrary.

Example 3: subset of (a|b|c)* in which some letter appears at least 3 times.

RE (one possibility):
(a|b|c)* ((a|b)*c(a|b)*c(a|b)*c(a|b)*
| (c|b)*a(c|b)*a(c|b)*a(c|b)*
| (a|c)*b(a|c)*b(a|c)*b(a|c)*) (a|b|c)*
exists an 8-state NFA
minimal DFA has 28 states

See the PLP Companion Site, Sec. 2.4.1

That NFA, of course, doesn't come from the standard construction.
Are there any that *do*, and for which the DFA is bigger yet?
Absolutely!

Example 4: subset of (0|1|2|3|4|5|6|7|8|9)* in which some digit appears at least 10 times.

minimal DFA has 10,000,000,001 states
one RE is

```
(0|1|2|3|4|5|6|7|8|9)*
(
   ((1|2|3|4|5|6|7|8|9)* 0 (1|2|3|4|5|6|7|8|9)* 0
    (1|2|3|4|5|6|7|8|9)* 0 (1|2|3|4|5|6|7|8|9)* 0
    (1|2|3|4|5|6|7|8|9)* 0 (1|2|3|4|5|6|7|8|9)* 0
    (1|2|3|4|5|6|7|8|9)* 0 (1|2|3|4|5|6|7|8|9)* 0
    (1|2|3|4|5|6|7|8|9)* 0 (1|2|3|4|5|6|7|8|9)* 0 (1|2|3|4|5|6|7|8|9)*)
 |
   ((0|2|3|4|5|6|7|8|9)* 1 (0|2|3|4|5|6|7|8|9)* 1
    (0|2|3|4|5|6|7|8|9)* 1 (0|2|3|4|5|6|7|8|9)* 1
    (0|2|3|4|5|6|7|8|9)* 1 (0|2|3|4|5|6|7|8|9)* 1
    (0|2|3|4|5|6|7|8|9)* 1 (0|2|3|4|5|6|7|8|9)* 1
    (0|2|3|4|5|6|7|8|9)* 1 (0|2|3|4|5|6|7|8|9)* 1 (0|2|3|4|5|6|7|8|9)*)
 | ...
 |
   ((0|1|2|3|4|5|6|7|8)* 9 (0|1|2|3|4|5|6|7|8)* 9
    (0|1|2|3|4|5|6|7|8)* 9 (0|1|2|3|4|5|6|7|8)* 9
    (0|1|2|3|4|5|6|7|8)* 9 (0|1|2|3|4|5|6|7|8)* 9
    (0|1|2|3|4|5|6|7|8)* 9 (0|1|2|3|4|5|6|7|8)* 9
    (0|1|2|3|4|5|6|7|8)* 9 (0|1|2|3|4|5|6|7|8)* 9 (0|1|2|3|4|5|6|7|8)*)
)
(0|1|2|3|4|5|6|7|8|9)*
```

anybody believe the automatically constructed NFA for that has 10 billion states?

We can also build a RE from a DFA (as shown in Section 2.4.1 on the PLP companion site). This completes a proof that the two notations are equally powerful. Nobody does this in practice, however; it amounts to converting the computer-friendly notation into a human-friendly notation, and we usually want to go the other way.

In a real compiler, the input may have errors, including lexical errors.

Consider, in Java or C, the input

```
int myVaria$ble
```

That dollar sign isn't supposed to be there.

What should a scanner do?

Generally suffices to
   return the longest token that starts at the beginning of the
      remaining part of the input program
      (after deleting white space and comments)
   or, if there isn't a valid token starting there, delete characters
      until there is, print an error message reporting the deletion,
      and return the token found

Pseudocode for a table-driver scanner can be found in the text.

==========
CONTEXT FREE GRAMMARS

Here's a grammar for a simple desk calculator language
(from the intro lecture notes):

1	*pgm*	⟶	*stmt_list* `$$`
2	*stmt_list*	⟶	*stmt_list stmt* | ε
3	*stmt*	⟶	`id :=` *expr* | `read id` | `write` *expr*
4	*expr*	⟶	*term* | *expr add_op term*
5	*term*	⟶	*factor* | *term mult_op factor*
6	*factor*	⟶	`(` *expr* `)` | `id` | `lit`
7	*add_op*	⟶	`+` | `–`
8	*mult_op*	⟶	`*` | `/`

[This happens to be a “bottom-up” grammar -- one of the two kinds that are easy to parse.]

Terminology:
   CF grammar
   symbols
      terminals (tokens)
      non-terminals
   start symbol
   production
   derivation (see example below)

left-most
right-most (canonical)
sentential form

[Useless symbols: non-terminals that can't derive a token string, or tokens that can't be derived. We will assume we have none of these. They can be detected and removed automatically and efficiently.]

Consider the program

```
read A
read B
sum := A + B
write sum
write sum / 2
```

Derivation using the above grammar:

program
stmt_list $$
stmt_list *stmt* $$
stmt_list write *expr* $$
stmt_list write *term* $$
stmt_list write *term* *mult_op* *factor* $$
stmt_list write *term* *mult_op* lit $$
stmt_list write *term* / lit $$
stmt_list write *factor* / lit $$
stmt_list write id / lit $$
stmt_list *stmt* write id / lit $$
stmt_list write *expr* write id / lit $$
stmt_list write *term* write id / lit $$
stmt_list write *factor* write id / lit $$
stmt_list write id write id / lit $$
stmt_list *stmt* write id write id / lit $$
stmt_list id := *expr* write id write id / lit $$
stmt_list id := *expr* *add_op* *term* write id write id / lit $$
stmt_list id := *expr* *add_op* *factor* write id write id / lit $$
stmt_list id := *expr* *add_op* id write id write id / lit $$
stmt_list id := *expr* + id write id write id / lit $$
stmt_list id := *term* + id write id write id / lit $$
stmt_list id := *factor* + id write id write id / lit $$

stmt_list `id := id + id write id write id / lit $$`
stmt_list stmt `id := id + id write id write id / lit $$`
stmt_list `read id id := id + id write id write id / lit $$`
stmt_list stmt `read id id := id + id write id write id / lit $$`
stmt_list `read id read id id := id + id write id write id / lit $$`
`read id read id id := id + id write id write id / lit $$`

Each line is a sentential form. By definition that's a string of grammar symbols that occurs in the derivation of some string of terminals from the start symbol.

This is a "canonical" (right-most) derivation: at each step we have expanded the right-most non-terminal in the current sentential form. So each line is a "right sentential form."

Bottom-up parsers that read their input left-to-right to discover right-most derivations.
Top-down parsers that read their input left-to-right discover left-most derivations.

A Little Theory

A context-free grammar (CFG) is a *generator* for a CF language.
A parser is a language *recognizer*.

There is an infinite number of grammars for every context-free language.
But not all grammars are equal!

For any CFG we can create a parser that runs in O(n^3) time.
Early's algorithm (~emulation of an NPDA)
Cocke-Younger-Kasami (CYK) algorithm (dynamic programming)

O(n^3) time is clearly unacceptable for a parser in a compiler.

There are large classes of grammars for which we can build parsers that run in linear time. The two most important classes are called LL and LR.

LL stands for 'Left-to-right, Leftmost derivation'.
LR stands for 'Left-to-right, Rightmost derivation'.

We'll focus on LL parsing, which is what you're going to be using in your next assignment. The LR class is larger, but
- most programming languages have LL grammars
  (or something close enough to use with a couple hacks)
- LL parsing is generally simpler and easier to understand.

You commonly see LL or LR (or whatever) written with a number in parentheses after it. This number indicates how many tokens of look-ahead are required in order to parse. Most but not all real compilers use one token of lookahead.

Some compilers (e.g., for Fortran) have hacks to get more lookahead in special cases.

The open-source compiler-compiler ANTLR is LL(k).

LL parsers are also called 'top-down', or 'predictive' parsers.
LR parsers are also called 'bottom-up', or 'shift-reduce' parsers.
More on this in the next lecture.

There are several important sub-classes of LR parsers, including SLR and LALR. See Sec. 2.3.4 in the text (unassigned) if you're curious.

[

- Every LL(1) grammar is also LR(1), though right recursion in productions (analogous to left recursion, discussed in more detail in the next lecture) tends to require very deep stacks and complicates semantic analysis.
- Most but not all LL(1) grammars are also LALR(1).
- Every CF *language* that can be parsed deterministically has an SLR(1) grammar (which is automatically LALR(1) and LR(1)).
- Every deterministic CFL with the “prefix property” (no valid string is a prefix of another valid string -- every language augmented with an end-of-file marker fits the bill) has an LR(0) grammar, but it's almost certainly too ugly to use.

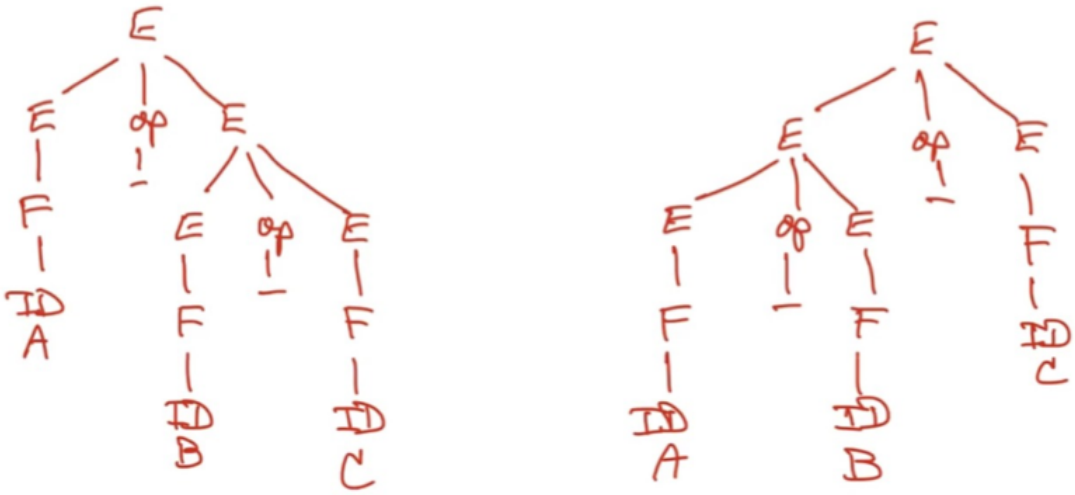
]

What makes a grammar “nice”?

It's particularly important that it be ***unambiguous*** -- no two parse trees for the same string. Consider what would have happened if bottom-up productions 4 and 5 were

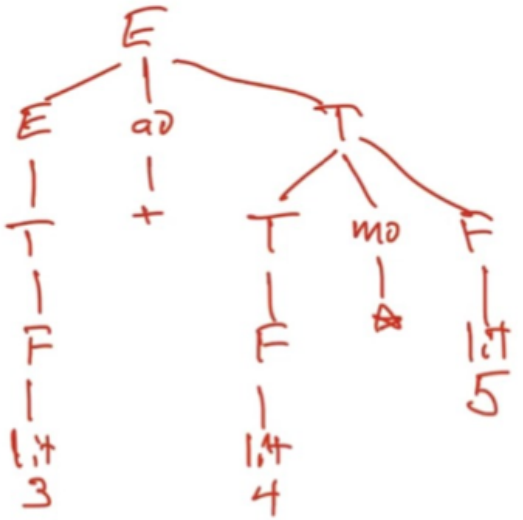
expr ⟶ *factor* | *expr op expr*

This gives us two parse trees for A – B – C :



Also nice if the parse trees reflect semantic structure, but that's not essential. Our bottom-up calculator grammar nicely captures the notion of precedence:

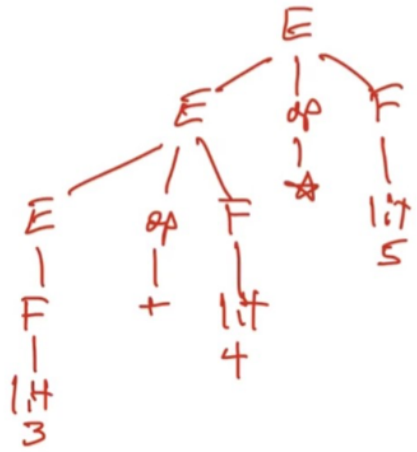
Here's a bottom-up parse tree for 3 + 4 ∗ 5 :



Consider what would have happened if productions 4 and 5 in the bottom-up grammar were

expr ⟶ *factor* | *expr op factor*

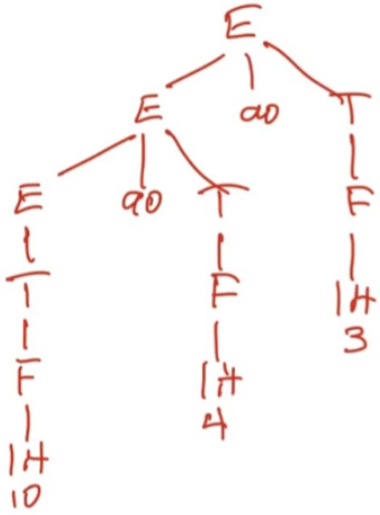
This gives us a different parse tree for `3 + 4 * 5` :



There is nothing *wrong* with this grammar or this tree, but you can see why the first one might be easier to translate into a syntax tree.

Our grammar also captures the notion of left associativity:

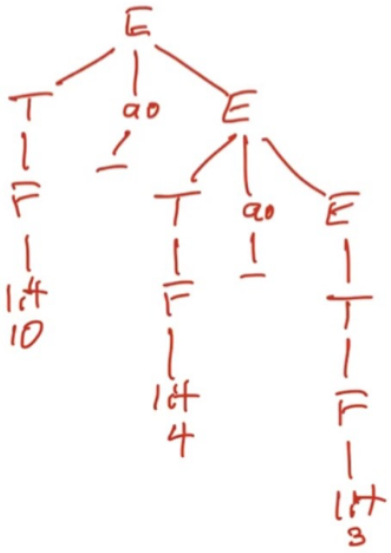
Here's a bottom-up parse tree for `10 - 4 - 3` :



Consider what would have happened if production 4 was

expr ⟶ *term* | *term add_op expr*

This gives us a different parse tree for `10 – 4 – 3` :



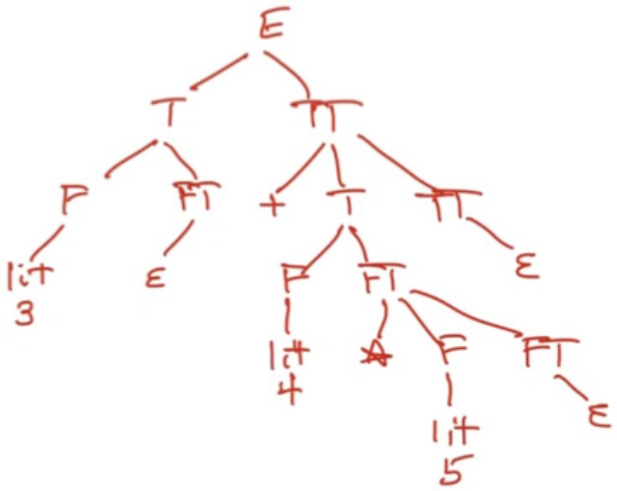
Again, there is nothing *wrong* with this grammar or this tree, but you can see why the first one might be easier to translate into a syntax tree.

Here is an LL(1) (top-down) grammar for the same language:

1	*pgm*	⟶	*stmt_list* `$$`
2	*stmt_list*	⟶	*stmt stmt_list* | ε
3	*stmt*	⟶	`id :=` *expr* | `read id` | `write` *expr*
4	*expr*	⟶	*term term_tail*
5	*term_tail*	⟶	*add_op term term*_tail | ε
6	*term*	⟶	*factor fact_tail*
7	*fact_tail*	⟶	*mult_op factor fact*_tail | ε
8	*factor*	⟶	`(` *expr* `)` | `id` | `lit`
9	*add_op*	⟶	`+` | `–`
10	*mult_op*	⟶	`*` | `/`

Like the bottom-up grammar, the top-down one captures precedence, but most people don't find it as pretty. Operands of a given operator aren't in a RHS together, and the resulting parse trees look a bit strange.

Here's a top-down parse tree for `3 + 4 * 5` :



It still seems to suggest that multiplication groups more tightly than addition, but in a lopsided sort of way.

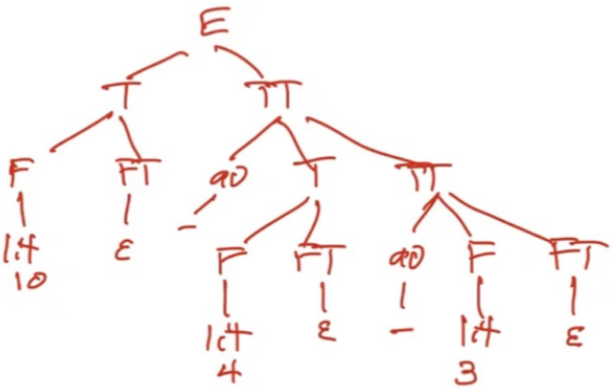
The simplicity of the parsing algorithm makes up for this weakness, in my opinion. As we'll see later, top-down parsing also makes it easier to handle special cases and to produce good error messages when the input program has syntax errors.

gcc switched from bottom-up to top-down parsing around 2005

LLVM's clang front end also uses top-down parsing

Also note that the top-down grammar doesn't capture associativity: in order to parse top-down left-to-right, we end up with a tree that tends to associate to the right.

Here's a top-down parse tree for `10 - 4 - 3` :



There's no getting around this in the parser. Have to take care (by hand) to make sure that the *syntax tree* reflects associativity correctly.

==========

TOP-DOWN AND BOTTOM-UP PARSING

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

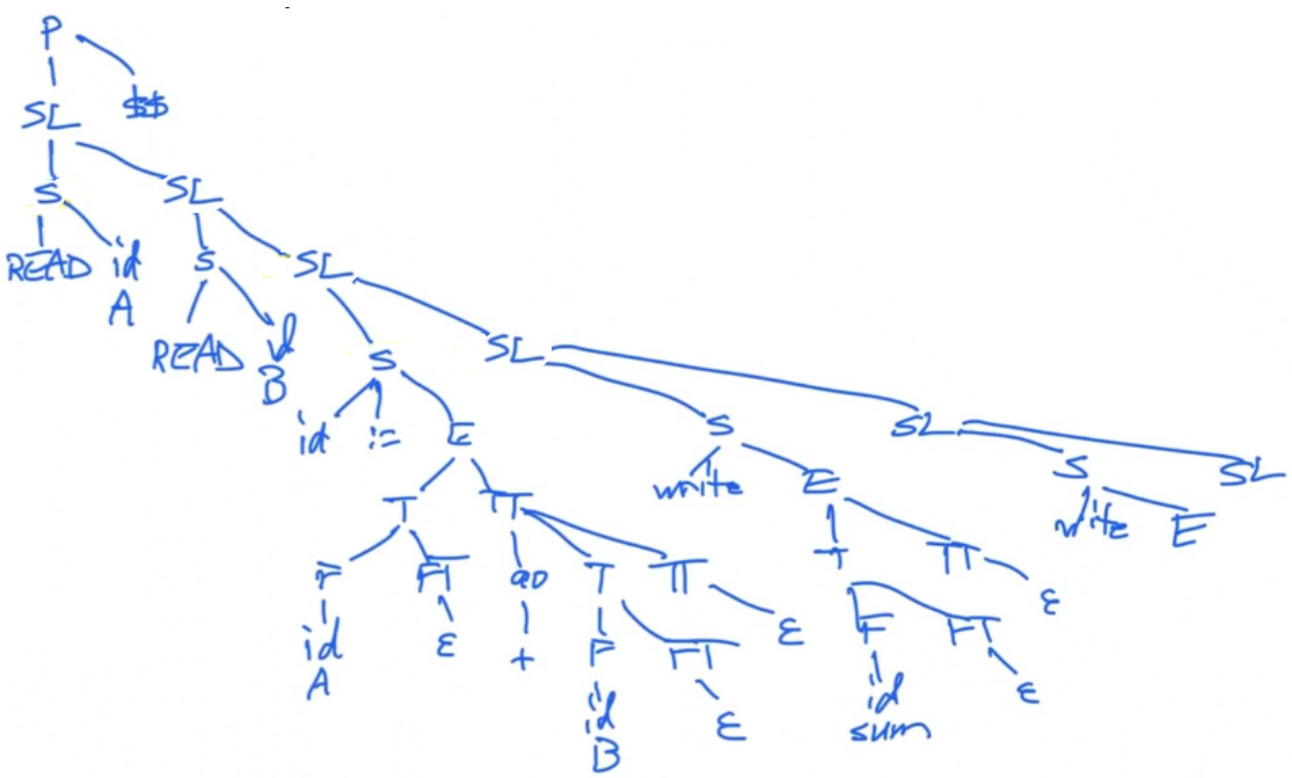
How do we parse a string with the top-down grammar? You can get the general idea by building the parse tree incrementally by hand:

> 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)

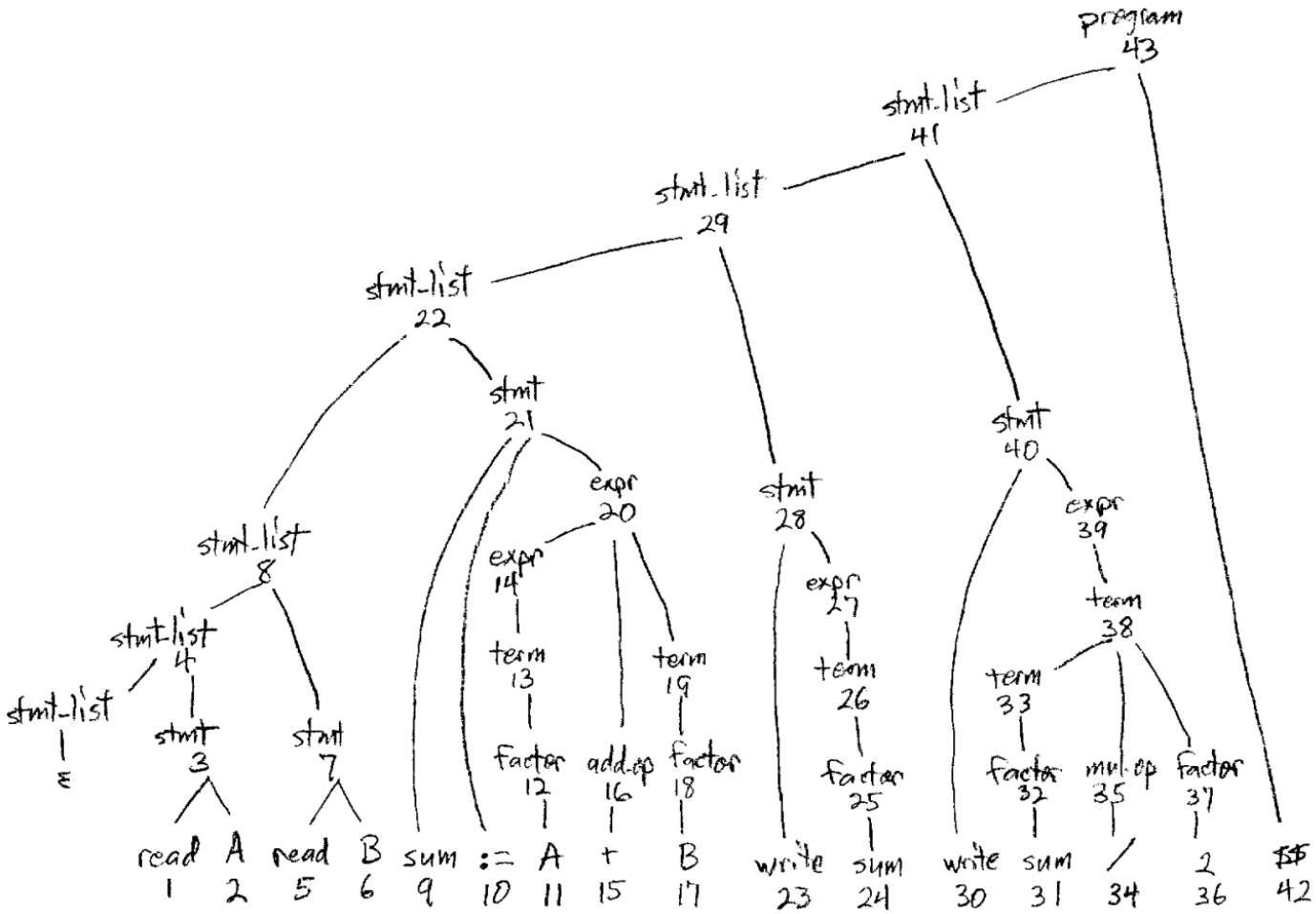
A bottom-up parser has a complex state machine.
- uses current state to make decisions
- I won't be showing you the state machine in this example

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

```
read A
read B
sum := A + B
write sum
write sum / 2
```

P → *SL* $$
SL → *SL S* | ε
S → id := *E* | read id | write E
E → *T* | *E ao T*
T → *F* | *T mo F*
F → (*E*) | id | lit
ao → + | −
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.

NB: I claimed earlier that a bottom-up parser discovers a rightmost derivation. We can see that in the example above. The parser starts with the last line of the derivation (the string of tokens). It then repeatedly finds the *previous line*, which glues together the leftmost not-glued together stuff. That means the *top* lines of the derivation, once discovered, are expanding the *rightmost* stuff.

==========

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 subroutines 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:

$$\text{FIRST}(\alpha) \equiv \{c : \alpha \Rightarrow^* c\ \beta\}$$
$$\text{FOLLOW}(A) \equiv \{c : S \Rightarrow^+ \alpha\ A\ c\ \beta\}$$
$$\text{PREDICT}(A \rightarrow \alpha) \equiv \text{FIRST}(\alpha) \cup (\text{if } \alpha \Rightarrow^* \varepsilon \text{ then FOLLOW}(A) \text{ else } \emptyset)$$

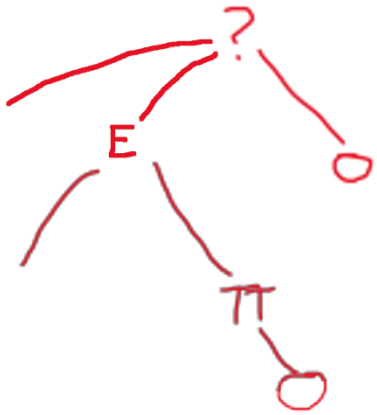
Here $\rightarrow$ is the familiar "goes to" symbol used in productions.
$\Rightarrow$ means "derives" — can be replaced by. Note that its LHS doesn't have to be a single symbol.
$\Rightarrow^*$ means "derives in zero or more steps"
$\Rightarrow^+$ means "derives in one or more steps"

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.

FIRST sets capture the "RHS can start with 'c'" case.
FOLLOW sets capture the "RHS can generate ε" case.
Consider, for example, a TT node of an under-construction parse tree in our calculator language.



We should predict TT → ao T TT if the upcoming token from the scanner is a + or −, which can go in the lower circle in the pic—that's the FIRST case. It can happen if the question mark is S and the stuff to the left is "write" or "id :=".

We should predict TT → ε if the upcoming token from the scanner is a ')', which can go in the other circle (the first leaf *to the right of* the TT—that's the FOLLOW case. It can happen if the question mark is F and the stuff to the left is "(".

The calculator language is simple enough that one can figure out PREDICT sets 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 ε, 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 A < B then
    if C < D then                        if C < D then
        X := 1           or                  X := 1
    else                                 else
        X := 2                               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 second 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
```

==========

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 ∈ FIRST(stmt)
                stmt()      -- try again
                return
            elsif input_token ∈ (FOLLOW(stmt) U {$$})
                return      -- caller can probably make progress
            else get_next_token()
                -- get_next_token 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 ∈ (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.

In the general case, context-sensitive FOLLOW sets are surprisingly easy to compute. The augmenting production passes {$$} to the routine for the start symbol (e.g., program). When calling a routine in the middle of an arm of a switch statement, we pass FIRST(α), where α comprises the symbols remaining to be parsed in this arm of the switch. If $\alpha \Rightarrow^* \varepsilon$, then we pass FIRST(α) unioned with the context-specific FOLLOW set we ourselves were passed.

In the example above, early detection of the error allows the parser to, effectively, "correct" the input into

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

That's not "right", but it's certainly better than what we got with delayed detection.

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

```
procedure check_for_error(sym, CSFset)
    if not (input_token ∈ FIRST(sym)
            or (sym ⇒+ ε and input_token ∈ CSFset)
        report_error()
        repeat
            get_next_token()
        until input_token ∈ (FIRST(sym) U CSFset 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.

==========

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 | E
                       stmt          → ID := expr | READ ID | WRITE expr
                       expr          → term term_tail
program:               term_tail     → add_op term term_tail | E
    read A             term          → factor fact_tail
    read B             fact_tail     → mult_op factor fact_tail | E
    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 + ...
```

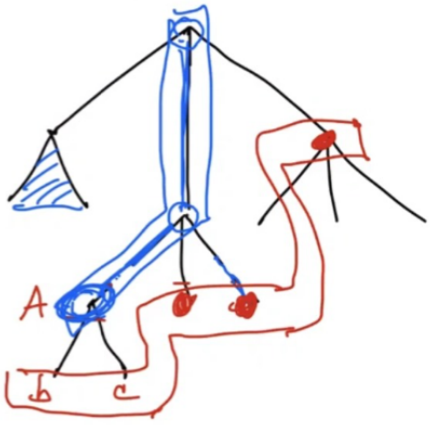
```
stmt_list $$                                  read B sum := A + ...
stmt stmt_list $$                             read B sum := A + ...
READ ID stmt_list $$                          B sum := A + B ...
ID stmt_list $$                               sum := A + B write ...
stmt_list $$                                  sum := A + B write ...
stmt stmt_list $$                             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
    (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 $\Rightarrow^*$ E then true else false
FIRST(α) == {c : α $\Rightarrow^*$ c β}
FOLLOW(A) == {c : S $\Rightarrow^+$ α A c β}
PREDICT(A $\rightarrow$ α) == FIRST(α) U (if EPS(α) then FOLLOW(A) ELSE ∅)

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 $\rightarrow$ E, then EPS(A) = true
c in FIRST(c)

How to deduce?
If EPS(α) = true and A $\rightarrow$ α, then EPS(A) = true
If A $\rightarrow$ B β, then FIRST(A) $\supset$ FIRST(B)
If A $\rightarrow$ α B β, then FOLLOW(B) $\supset$ FIRST(β)
If A $\rightarrow$ α B (or A $\rightarrow$ α B β and EPS(β) = true)
  then FOLLOW(B) $\supset$ FOLLOW(A)

This last one is tricky. It's *not* true the other way around.
That is, A $\rightarrow$ α B does *not* imply that FOLLOW(A) $\supset$ 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
(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.