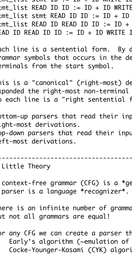
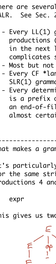
```
=========================================
CONTEXT FREE GRAMMARS

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

    1   program       ->  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 | LITERAL
    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 $$
stmt_list WRITE term mult_op LITERAL $$
stmt_list WRITE term / LITERAL $$
stmt_list WRITE factor / LITERAL $$
stmt_list WRITE ID / LITERAL $$
stmt_list WRITE ID / LITERAL $$
stmt_list stmt WRITE ID / LITERAL $$
stmt_list WRITE expr WRITE ID / LITERAL $$
stmt_list WRITE term WRITE ID / LITERAL $$
stmt_list WRITE factor WRITE ID / LITERAL $$
stmt_list WRITE ID WRITE ID / LITERAL $$
stmt_list stmt WRITE ID WRITE ID / LITERAL $$
stmt_list ID := expr WRITE ID WRITE ID / LITERAL $$
stmt_list ID := expr add_op term WRITE ID WRITE ID / LITERAL $$
stmt_list ID := expr add_op factor WRITE ID WRITE ID / LITERAL $$
stmt_list ID := expr add_op ID WRITE ID WRITE ID / LITERAL $$
stmt_list ID := expr + ID WRITE ID WRITE ID / LITERAL $$
stmt_list ID := term + ID WRITE ID WRITE ID / LITERAL $$
stmt_list ID := factor + ID WRITE ID WRITE ID / LITERAL $$
stmt_list ID := ID + ID WRITE ID WRITE ID / LITERAL $$
stmt_list stmt ID := ID + ID WRITE ID WRITE ID / LITERAL $$
stmt_list READ ID ID := ID + ID WRITE ID WRITE ID / LITERAL $$
stmt_list READ ID READ ID ID := ID + ID WRITE ID WRITE ID / LITERAL $$
READ ID READ ID ID := ID + ID WRITE ID WRITE ID / LITERAL $$

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 factor

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 :
```



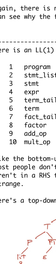
```
Consider what would have happened if productions 4 and 5 in the
bottom-up grammar were

    expr          ->  factor | expr op factor
```



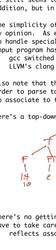
```
This gives us a different parse tree for 3 + 4 * 5 :
```

```
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   program       ->  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 | LITERAL
    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.


Next lecture: algorithms for top-down v bottom-up parsing.
```