CS131 Week 2

Week 2 Lecture 1

==========

*For this lecture, capital letters are nonterminal symbols, lowercase are terminal symbols

Grammars

- are almost programs
- are descriptions of languages
  - to generate sentences
  - to parse sentences

Problems with grammars:

- blind-alley rule: context-free grammar problem where no way to terminate, i.e. :
  - S -> aT
    S -> b
  - NOTICE: T is nonterminal symbol, no production for T, PROBLEM!

Also...

1. **nonterminal used but not defined (useless rule)**
2. **nonterminal defined but not used (useless nonterminal)**
3. **nonterminal defined but not reachable from start symbol (variation of #2)**
4. **nonterminals that can never produce string of ONLY terminals**
    1. infinitely recursing nonterminal
    2. interdependent nonterminals
        1. T -> Ua
           U -> Tb
5. **BIGGER PROBLEM: trying to use grammar to capture constraints that are best captured elsewhere**
    1. Example from English:
       _____

       S -> NP VP .
       NP -> The dog
       NP -> The cats
       VP -> eat food
       VP -> drinks water

       - Notice: the grammar can generate

The dog drinks water (Correct)
The cats eat food (Correct)
The dog eat food (Wrong)
The cats drinks water (Wrong)

Constraints better captured elsewhere:
S -> SNP SNP . (Singular noun phrase)
S -> PNP PNP . (Plural noun phrase)

BUT still need to capture more constraints, not just plural/singular
- number (plural/singular)
- tense (past present future)
- mood (not important, just an example)

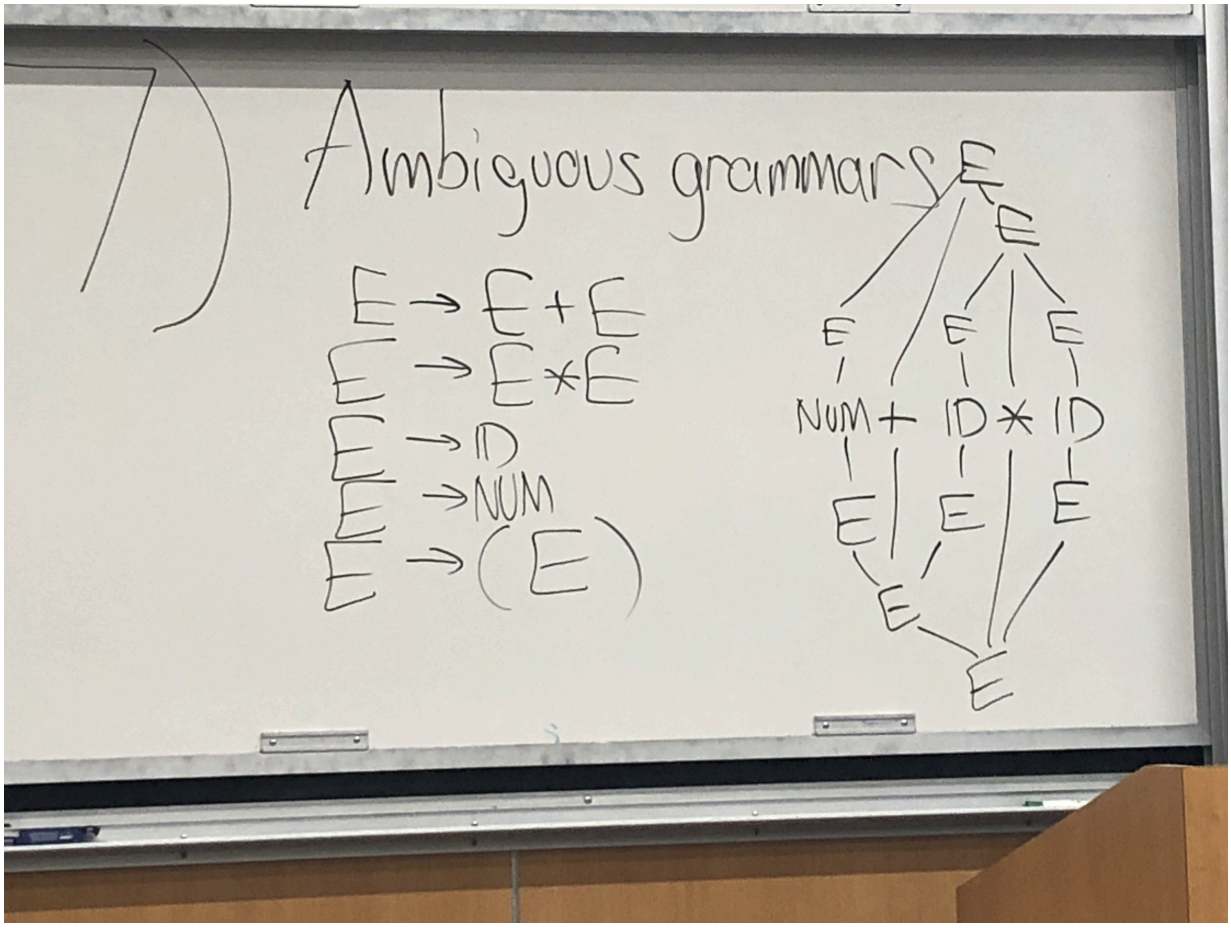
6. **Using grammars to specify details best handled by simpler technology**

   Example from C++:
   identifier -> a|b|c|...|z|A|B|C|...|Z
   identifier -> identifier(a|b|...|_|0|1|2|...|9)

   comment -> // commentbody \n'

   MIDTERM QUESTION: Write a grammar that only allows valid identifiers:
   HUGE hassle
   - Best if the lexer will perform this task, but some ppl say language inventors should specify in their grammar rather than lexer

   KEY TAKEAWAY: Some details are best handled by simpler territory
7. **BIGGEST PROBLEM: Ambiguous Grammar**

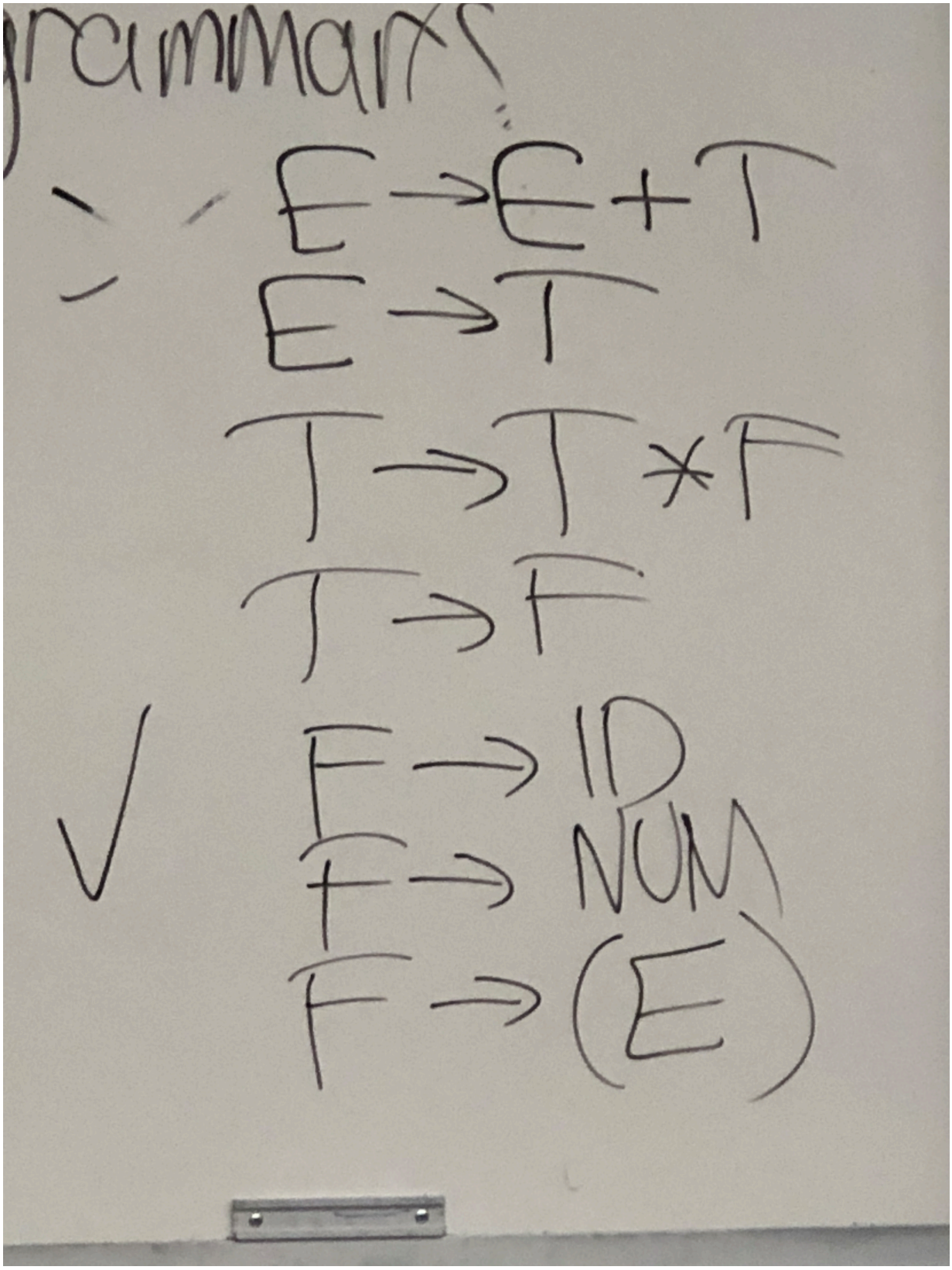


How to know if grammar is ambiguous?

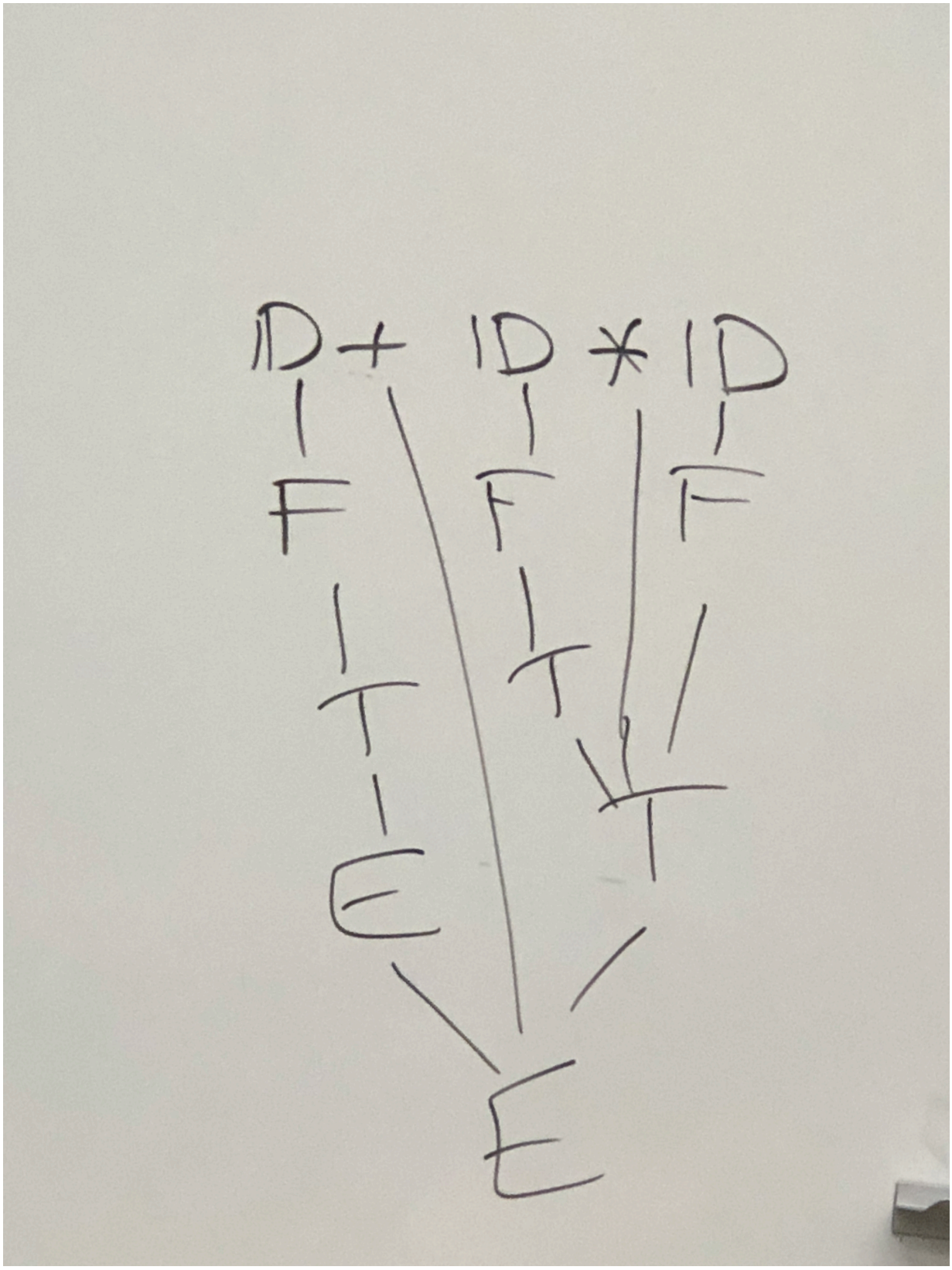
- You should start to get worried if you see a nonterminal on left, and nonterminal terminal nonterminal on the right
- **To show grammar is ambiguous, prove there is string with 2+ parse trees**

How to fix ambiguous grammar??

- **Complicate grammar** to force one single parse tree (force associativity rules), i.e.
  - force right associativity e.g.
    change above picture grammar (E -> E + E) to
    _____
    E -> E + T
    E -> T
    T -> ID
    T -> NUM
    T -> (E)
  - CONS operator in OCaml is right associative
    E -> T::E
    T ->
  - More examples



its parse tree:



Ambiguity continuation with Advanced Case Study of Syntax for C

Problem: make a syntax for C

Break down problem:
1. Find simplest statements in C
2.

Simplest statements in C/C++:
stmt ->
;
- empty statement

return;
- empty return

return expr;
- return something

expr;
- MOST COMMON STATEMENT

continue;
- continue statement

{ statement list }
- statement list in curly braces

Build up from there:
blocks ->
while (expr) stmt
do stmt while (expr);
if (expr) stmt
if (expr) stmt else stmt
switch (expr) stmt
...

Case study:
- return(n+1);
- Why do we use parentheses?
  - What if grammar is ambiguous?
  - With our current rules, we read as
  - **(return n) + 1;** <- WTF

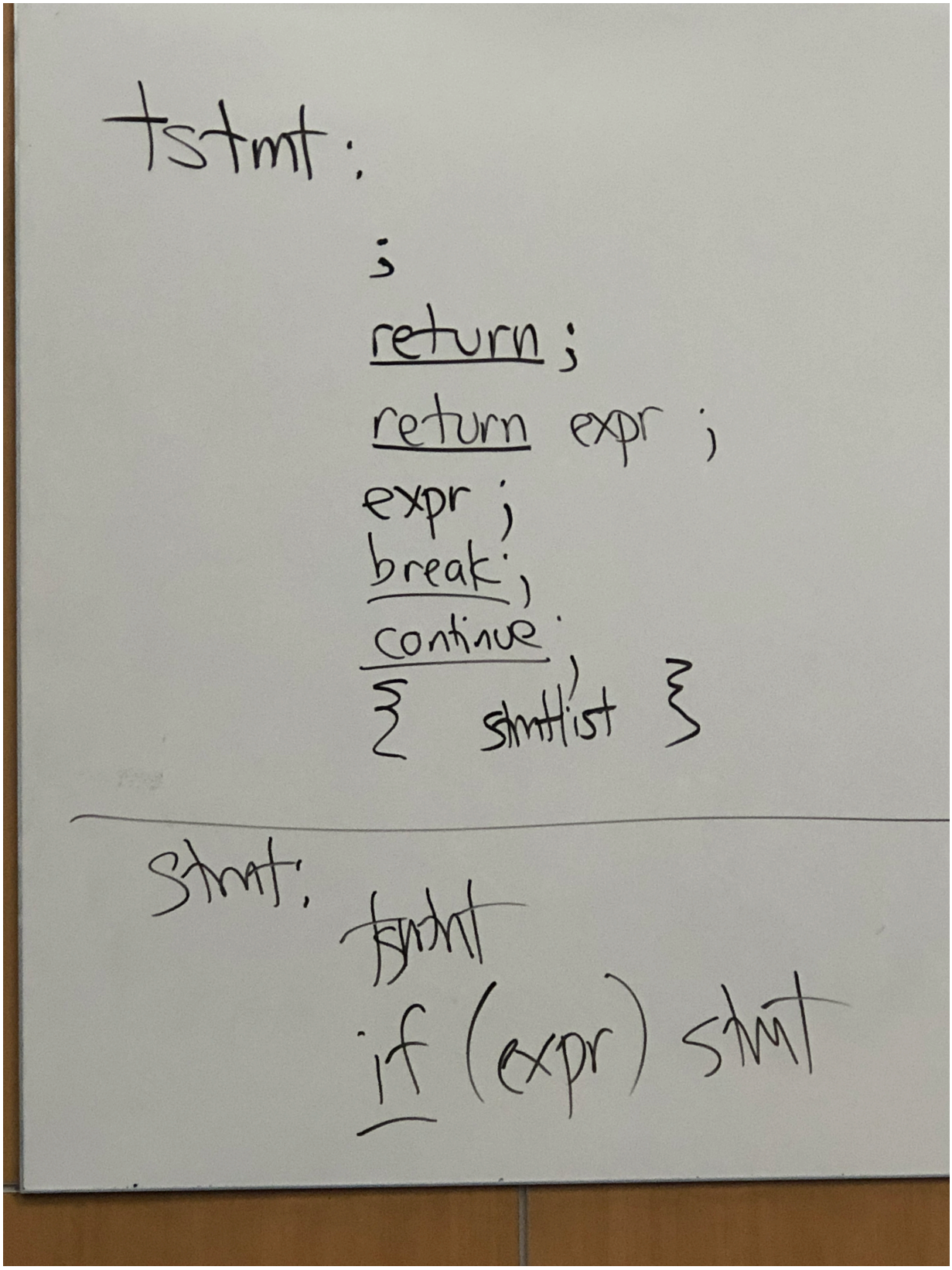
Sometimes, parentheses are necessary
e.g.
while (expr) stmt <- expr MUST have parenth, or expr merges with stmt
do stmt while (expr); <- doesn't need parenth, but C++ makers mandate parenth so ppl dont forget in the normal while statement

MAIN POINT OF EXAMPLE: there is ambiguity that needs to be resolved by some construct in grammar sometimes (like parentheses)

Big example of parsing ambiguity:
If else statement matching
———————————————

```
if (1) if (0) f(); else g();
       |———stmt————————|     (is the correct parse)
       |-stmt—|               (WRONG!)
```

To fix this ambiguity problem to force the first correct parse, we add nonterminal <tstmt>



How to know if your grammar has a problem?
- Experience

- Watch out for combinatorial explosion (if every time you add a rule you get a shit ton more possibilities, you probably have a problem)

Multiple Notations for Grammars
- BNF (Backus Naur Form)
- EBNF (extended BNF)
  - lots of different variations of EBNF

context-free grammars:
- nonterminal expansion doesn't depend on symbols around it
- i.e. left side of productions ONLY HAVE NONTERMINAL

nonterminal -> [symbol, symbol, symbol, symbol]
symbol -> nonterminal | terminal

Internet RFC 5322
EBNF:
*below is a regex expression for valid email msg ids
msg-id = "<" word *("." word) "@" atom *("." atom) ">"
                        ^^   ^   ^^   ^
BNF:
msg-id = "<" word dot words "@" atom dotatoms ">"
dotwords= ""
dotwords= "." word dotwords
dotatoms= ""
dotatoms= "." atom dotatoms

Message ID: <eggert$27f,"do"@cs.ucla.edu>

MIDTERM QUESTIONS: He'll probably ask to write BNF <-> EBNF

context sensitive grammars:
Sb -> aS
- how nonterminals are used depend on the symbols that are near it

Another example that is BNF
word = atom/quoted-string
atom = 1*<any CHAR except specials, SPACE, and CTLs> # in this stmt, * is regex, means 1 or more (should really be regex +)
- specials = ()<>@,;:\'".[]

- CTLs = '\000'-'\037', '\177'

quoted-string = <">*(qtext/quoted-pair)<">
qtext = <any CHAR except "\CR">
quoted-pair = "\"CHAR

ISO standard for EBNF
http://www.cl.cam.ac.uk/~mgk25/iso-14977.pdf might test on this?? very long
ISO-EBNF is essentially BNF with regex

EBNF	ISO-EBNF operators
[option]	X* (repetition)
{repetition}	X-Y (except)
(grouping)	X,Y (concatenate)
(*comment*)	X|Y (OR)
	= (definition (in a role))
	; (end of a role)

downwards is decreasing precedence

Syntax used to develop programming languages
syntax = syntax rule, {syntax rule};
syntax rule = meta id, '=', defns list, ';' ;
defns list = defn, { '|', defn } ;
...

—- 1/16/18

Week 2 Lecture 2

==========

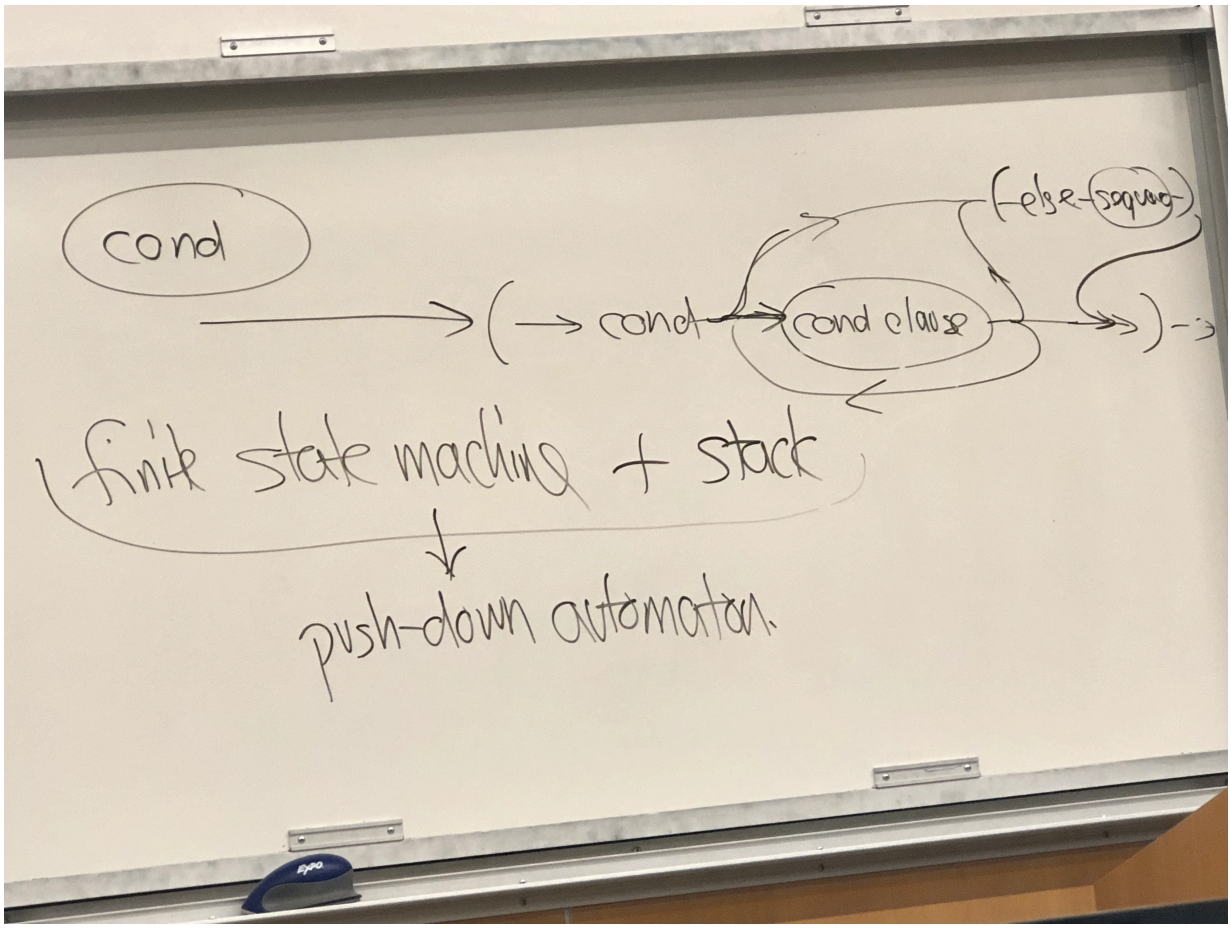
Syntax
- grammars
- BNF
- EBNF
- syntax diagrams

- generator
- recognizers/parsers

Scheme syntax for conditional expressions:
<cond> -> (cond <cond clause>+)
| (cond <cond clause>*

(else <sequence>))

Follows a FSM + stack (finite state machine that's recursive)
= push-down automaton



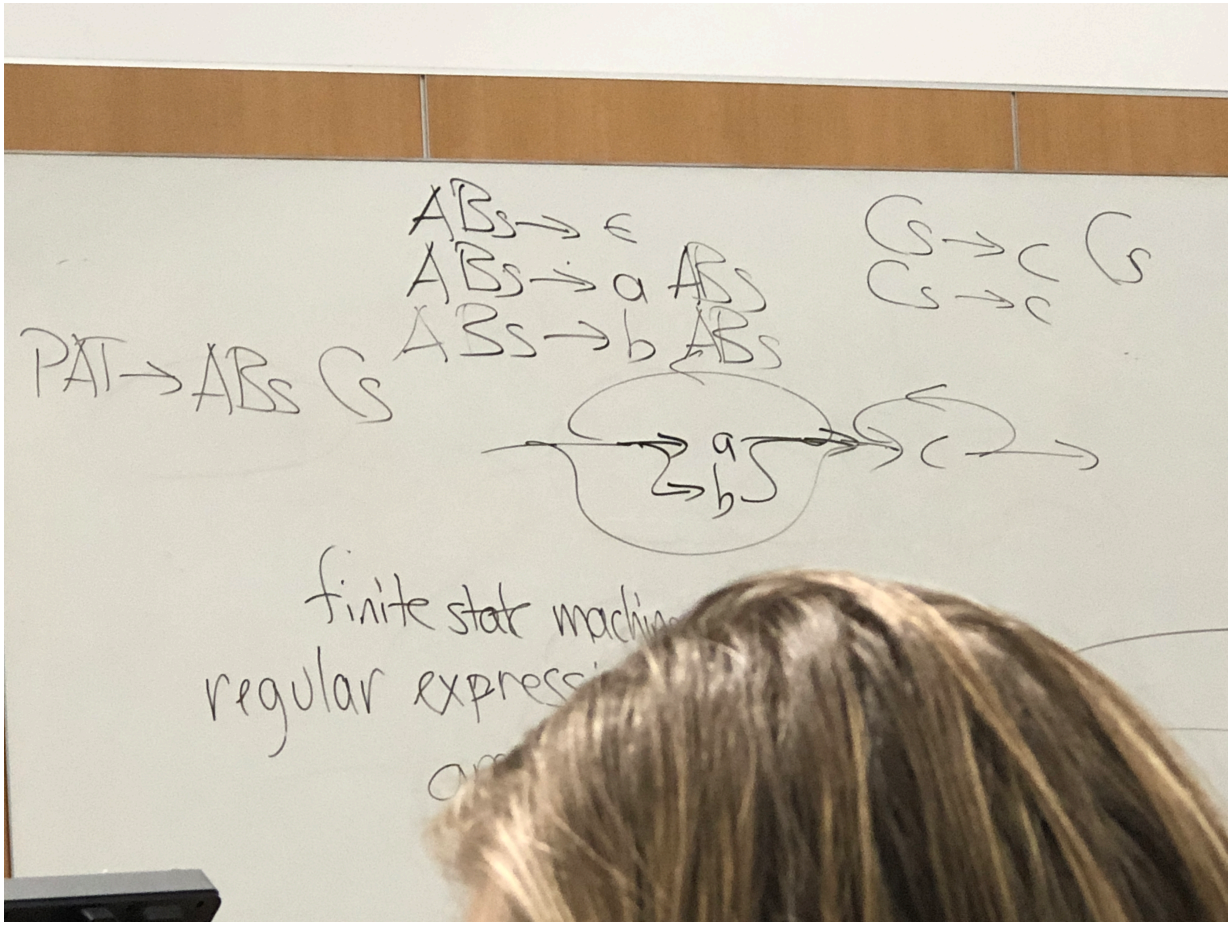

```
parse_cond() { <- recursive function, needs stack
        scan_for("(");
        scan_for("cond");
        if (lookahead(["("; "else"])) {
                scan_for("("); scan_for("else");
                parse_sequence();
                scan_for(")");
        } else
                while();
}
```

Some grammars don't need a stack, e.g. msg-id from last class - FSM without stack is enough

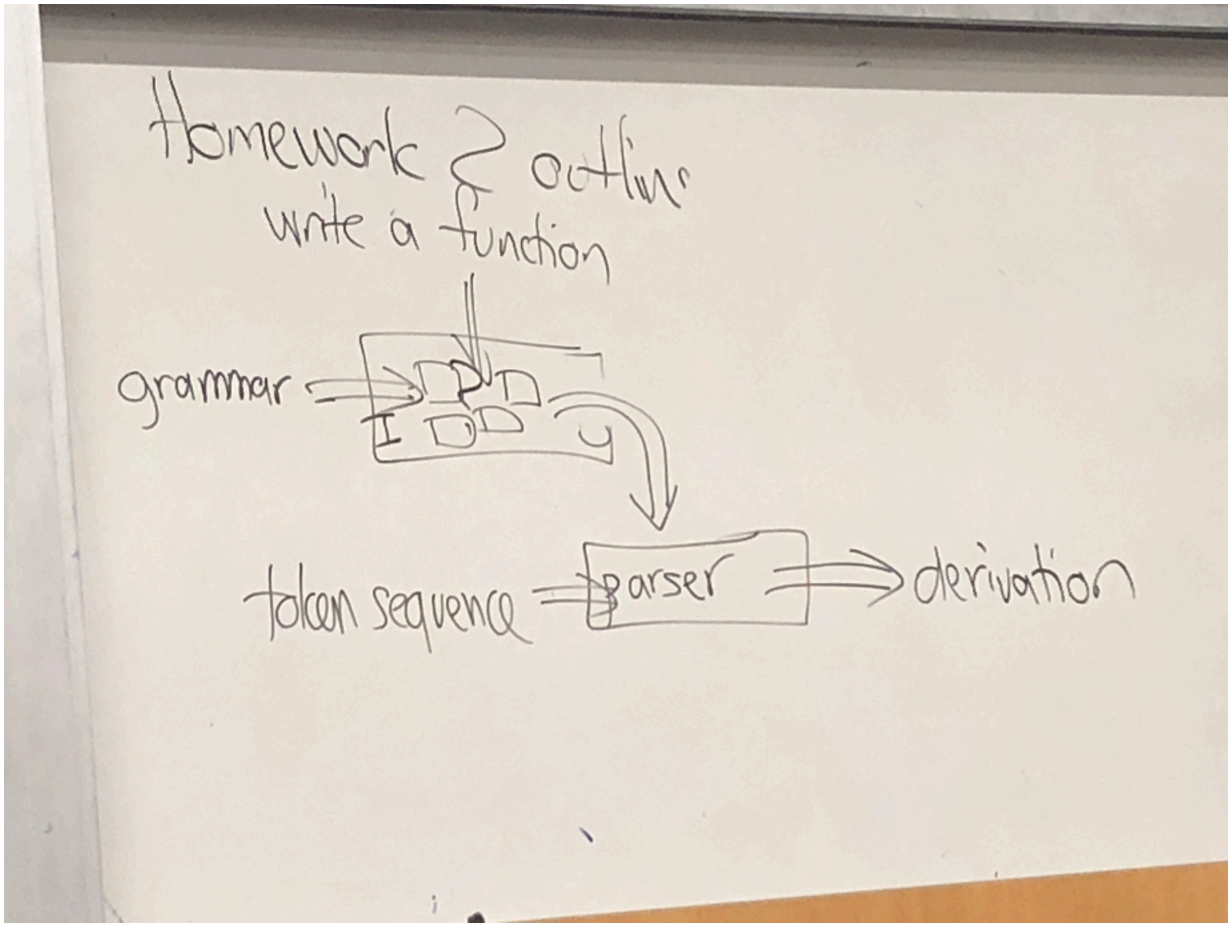
grep [pattern]

i.e. grep -E '(a|b)*c+'

Get statement into EBNF
First make a FSM



HW 2 Outline: Saying Functional Programming is really effective in Distributed Computing (taking over)



in functional programming, usually recursive programming will not just have 1 function: will call other recursive functions

- try to avoid exploding yourself (don't step through recursive functions that intercall each other)
- assume other recursive function you are calling works perfectly, even though it calls the function you're not even done yet
- "Just trust yourself"

We will essentially be writing a program that takes in grammar and outputs a parser
BIG PICTURE: 3 features we need to support

1. recursion
2. alternation (OR)
3. concatenation (easy for finite state machines)

Eggert wishes more operators were added into programming languages, i.e.

- a<>b = a < b | a > b
- (a <> b) != (a != b)

Expressions and their problems
- precedence
- associativity

both of the above are handled by grammars
- user-defined operators
- user-defined syntax
  - e.g. Eggert thinks do while was written incorrectly
  - Usually after while loop or some expression check, the expression value should be true, i.e. if (a==b), while(a==b) we know a=b. do while (a==b) is not like this, a!=b after the while statement
  - #define until(expr) = while(!expr)
- side-effects and their problems
  - int a, b, x;
    x = (a=1) + (b=2) + (a+b);
    ^ 1 ^ 2 ^ we don't know, since C doesn't define order of execution for statements within semicolon: could execute left to right, could execute right to left (uses old value of a and b)
  - in real life situation:
    double *sp = &stack[10000];
    #define PUSH(E) *—sp=(E)
    #define POP() *sp++
    switch(op) {
    case DIVIDE:
    PUSH(POP()/POP()); break; // looks good, but really expands to
    // *—sp = *sp++/*sp++;
    // C std says behavior is undefined since order of execution matters, gcc decided to say sp subtracts 1 adds 2, accumulates to add 1, and thus fucks up everything
    }

Prolog lets you define your own operators
:- op (500, yfx, [+,-]).
:- op (400, yfx, [*,/]).
:- op (200, fy, [+,-]).
:- op (700, xfx, [=, \=, ==, >==, ...])
^ non associative

Syntax

When designing programming languages, should use precedence because people are used to it

Gives arise to functional programming:

Functional Programming

- motivations:
  - clarity
    - centuries of experience
    - i = i+1 -> FALSE
    - escape order of evaluation problems
  - performance
    - escape from von Neumann bottleneck
      - allows multiple CPUs to be accessing RAM at the same time and perform multiple executions
- no assignment statements
  - NO SIDE EFFECTS
  - state of machine changes (in a register, or file pointer, reads, writes, etc all change machine state)

function: mapping from a domain to a range

- partial function: domain not entirely covered (not all elements are mapped)

functional forms: function whose domain or range includes functions

- summation(xi + yi) from 0<= i <=100
- in C, probably looks like
  int plus(i) {return x[i] + y[i]}
  SIGMA(plus, 0, 100)

— 1/18/18