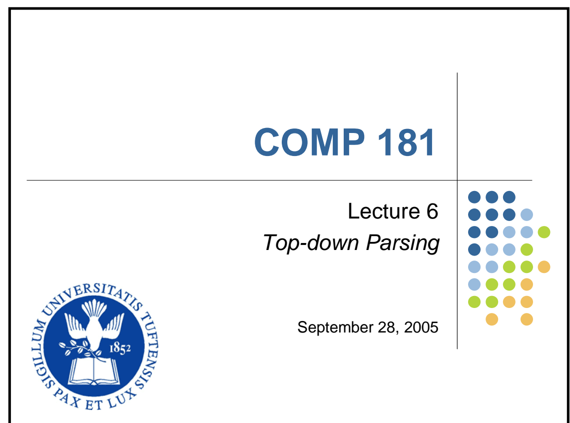
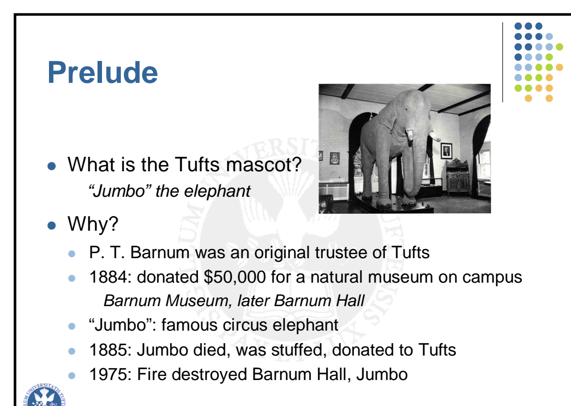
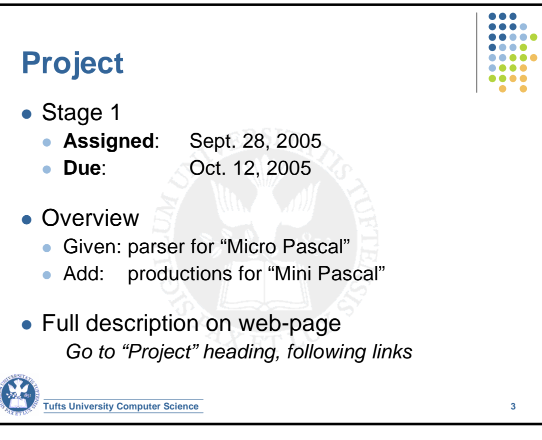
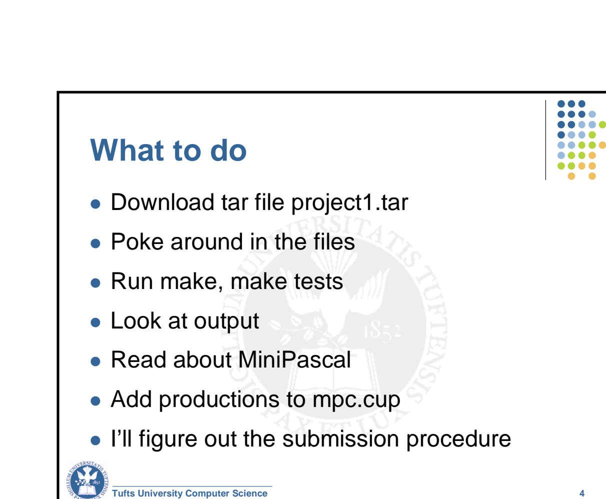
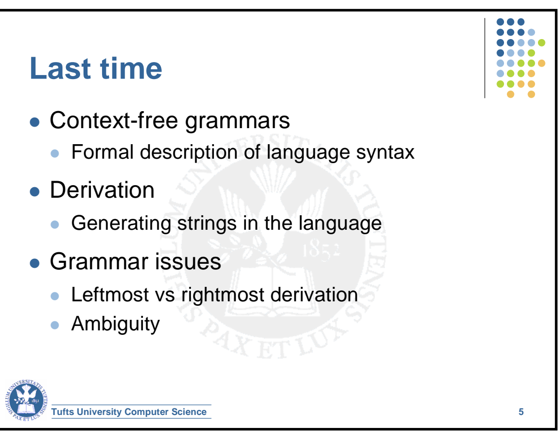
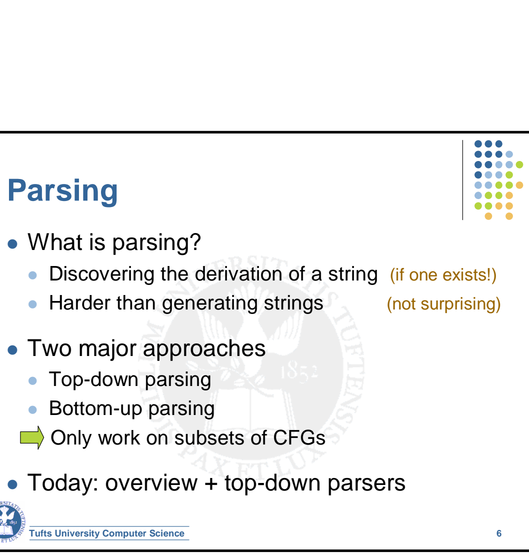

COMP 181
Lecture 6
Top-down Parsing
September 28, 2005
SIGILLUM UNIVERSITATIS TUFTENSIS
1852
PAX ET LUX

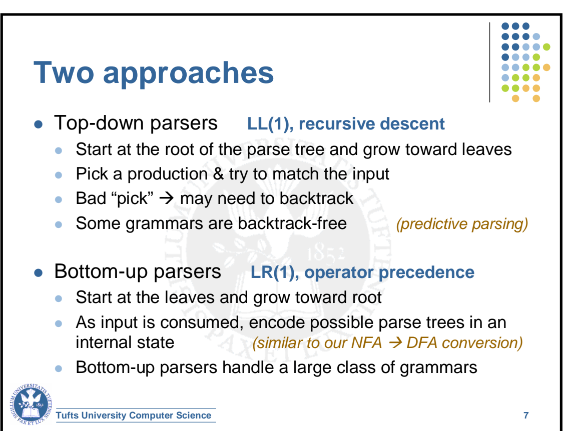

Prelude
What is the Tufts mascot?
"Jumbo" the elephant
Why?
P. T. Barnum was an original trustee of Tufts
1884: donated $50,000 for a natural museum on campus
Barnum Museum, later Barnum Hall
"Jumbo": famous circus elephant
1885: Jumbo died, was stuffed, donated to Tufts
1975: Fire destroyed Barnum Hall, Jumbo
Tufts University Computer Science
2

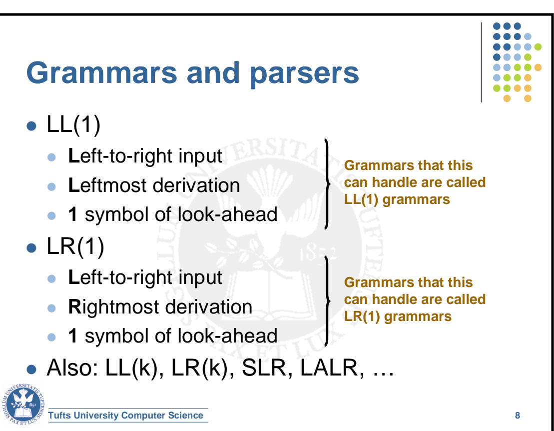

Project
Stage 1
Assigned: Sept. 28, 2005
Due: Oct. 12, 2005
Overview
Given: parser for "Micro Pascal"
Add: productions for "Mini Pascal"
Full description on web-page
Go to "Project" heading, following links
Tufts University Computer Science
3

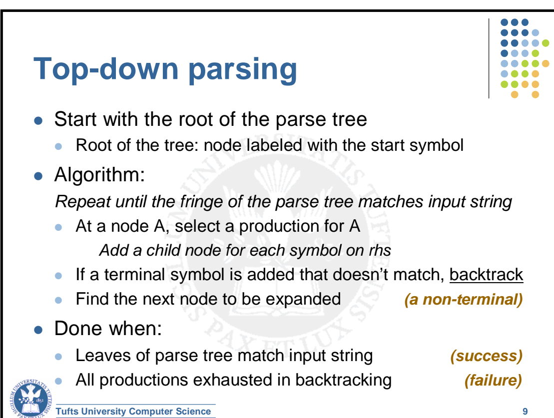

What to do
Download tar file project1.tar
Poke around in the files
Run make, make tests
Look at output
Read about MiniPascal
Add productions to mpc.cup
I'll figure out the submission procedure
Tufts University Computer Science
4

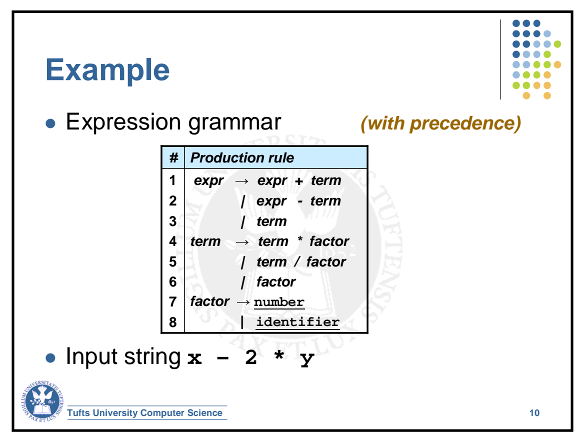

Last time
Context-free grammars
Formal description of language syntax
Derivation
Generating strings in the language
Grammar issues
Leftmost vs rightmost derivation
Ambiguity
Tufts University Computer Science
5

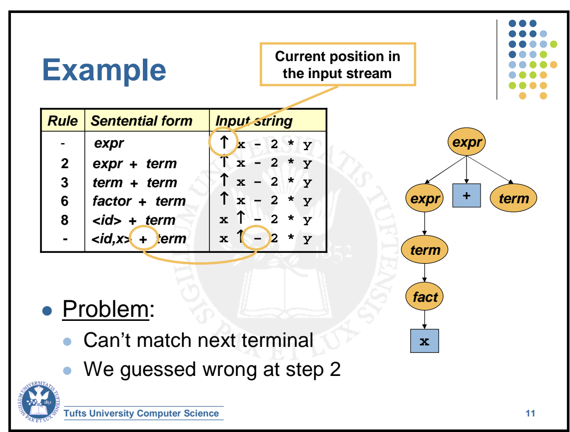

Parsing
What is parsing?
Discovering the derivation of a string (if one exists!)
Harder than generating strings (not surprising)
Two major approaches
Top-down parsing
Bottom-up parsing
Only work on subsets of CFGs
Today: overview + top-down parsers
Tufts University Computer Science
6

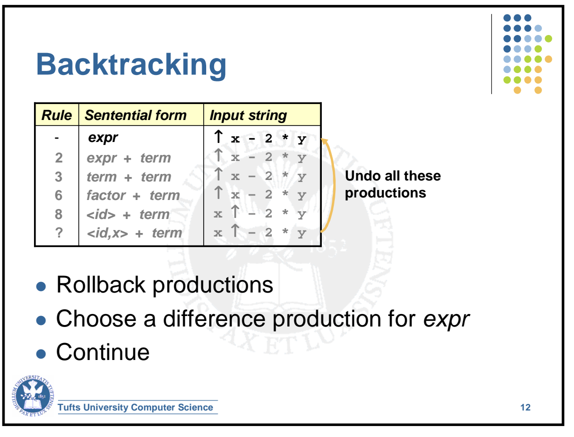

Two approaches
Top-down parsers
LL(1), recursive descent
Start at the root of the parse tree and grow toward leaves
Pick a production & try to match the input
Bad "pick" → may need to backtrack
Some grammars are backtrack-free
(predictive parsing)
Bottom-up parsers
LR(1), operator precedence
Start at the leaves and grow toward root
As input is consumed, encode possible parse trees in an internal state
(similar to our NFA → DFA conversion)
Bottom-up parsers handle a large class of grammars
Tufts University Computer Science
7

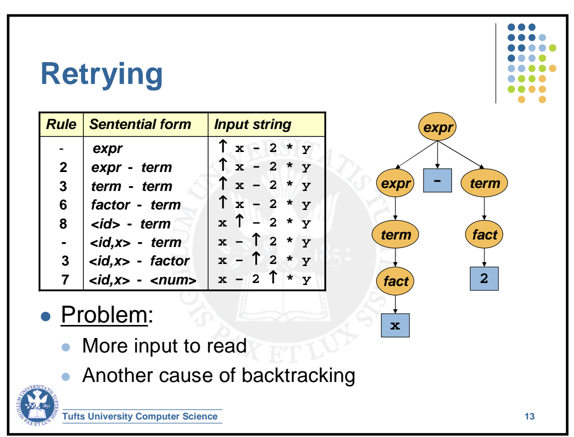

Grammars and parsers
LL(1)
Left-to-right input
Leftmost derivation
1 symbol of look-ahead
Grammars that this can handle are called LL(1) grammars
LR(1)
Left-to-right input
Rightmost derivation
1 symbol of look-ahead
Grammars that this can handle are called LR(1) grammars
Also: LL(k), LR(k), SLR, LALR, …
Tufts University Computer Science
8

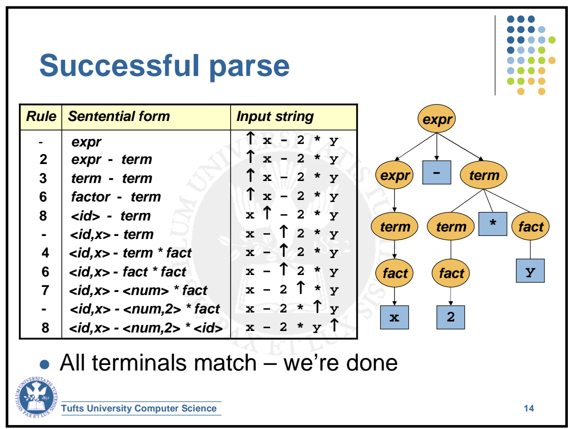

Top-down parsing
Start with the root of the parse tree
Root of the tree: node labeled with the start symbol
Algorithm:
Repeat until the fringe of the parse tree matches input string
At a node A, select a production for A
Add a child node for each symbol on rhs
If a terminal symbol is added that doesn't match, backtrack
Find the next node to be expanded
(a non-terminal)
Done when:
Leaves of parse tree match input string
(success)
All productions exhausted in backtracking
(failure)
Tufts University Computer Science
9

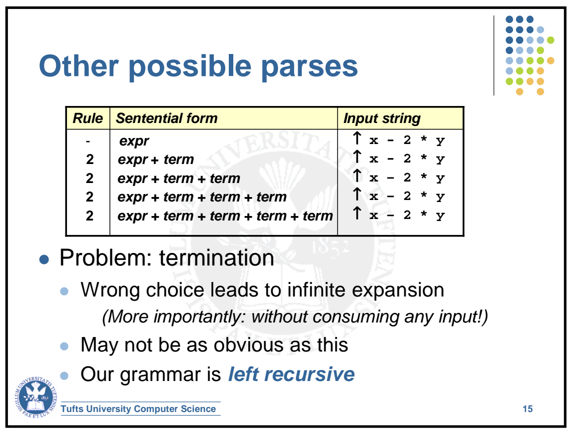

Example
Expression grammar
(with precedence)
Production rule
1 expr → expr + term
2 | expr - term
3 | term
4 term → term * factor
5 | term / factor
6 | factor
7 factor → number
8 | identifier
Input string x - 2 * y
Tufts University Computer Science
10

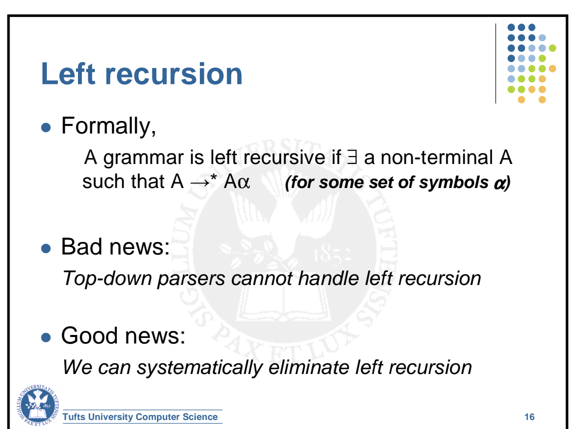

Example
Current position in the input stream
Rule Sentential form Input string
- expr ↑ x - 2 * y
2 expr + term ↑ x - 2 * y
3 term + term ↑ x - 2 * y
6 factor + term ↑ x - 2 * y
8 <id> + term x ↑ - 2 * y
- <id,x> + term x ↑ - 2 * y
expr
expr
+
term
term
fact
x
Problem:
Can't match next terminal
We guessed wrong at step 2
Tufts University Computer Science
11

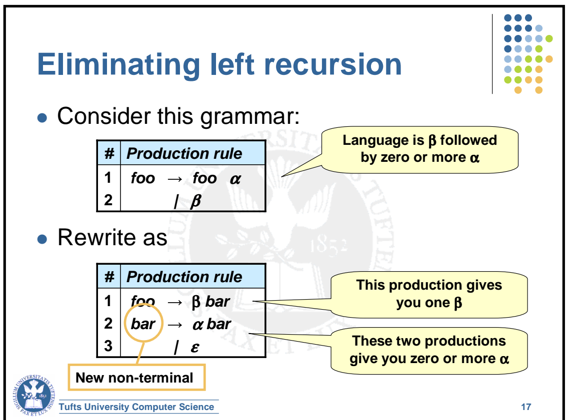

Backtracking
Rule Sentential form Input string
- expr ↑ x - 2 * y
2 expr + term ↑ x - 2 * y
3 term + term ↑ x - 2 * y
6 factor + term ↑ x - 2 * y
8 <id> + term x ↑ - 2 * y
? <id,x> + term x ↑ - 2 * y
Undo all these productions
Rollback productions
Choose a difference production for expr
Continue
Tufts University Computer Science
12

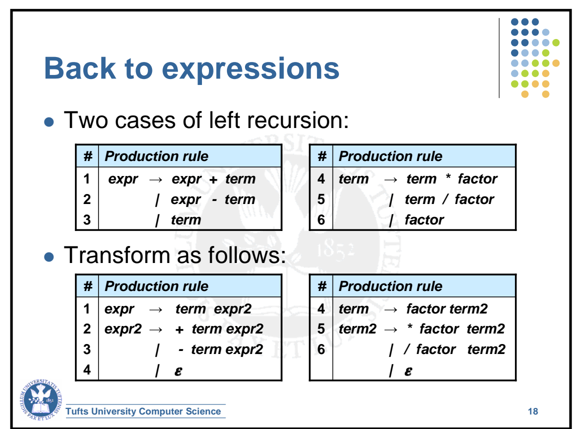

Retrying
Rule | Sentential form | Input string
- expr ↑ x - 2 * y
2 expr - term ↑ x - 2 * y
3 term - term ↑ x - 2 * y
6 factor - term ↑ x - 2 * y
8 <id> - term x ↑ - 2 * y
- <id,x> - term x - ↑ 2 * y
3 <id,x> - factor x - ↑ 2 * y
7 <id,x> - <num> x - 2 ↑ * y
expr
expr
-
term
term
fact
fact
2
x
Problem:
More input to read
Another cause of backtracking
Tufts University Computer Science
13

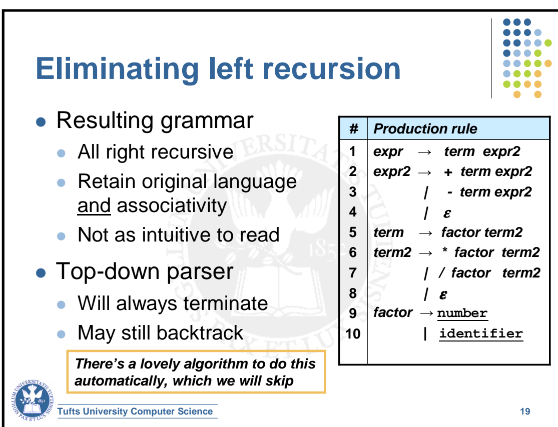

Successful parse
Rule | Sentential form | Input string
- expr ↑ x - 2 * y
2 expr - term ↑ x - 2 * y
3 term - term ↑ x - 2 * y
6 factor - term ↑ x - 2 * y
8 <id> - term x ↑ - 2 * y
- <id,x> - term x - ↑ 2 * y
4 <id,x> - term * fact x - ↑ 2 * y
6 <id,x> - fact * fact x - ↑ 2 * y
7 <id,x> - <num> * fact x - 2 ↑ * y
- <id,x> - <num,2> * fact x - 2 * ↑ y
8 <id,x> - <num,2> * <id> x - 2 * y ↑
expr
expr
-
term
term
term
*
fact
fact
fact
y
x
2
All terminals match – we're done
Tufts University Computer Science
14

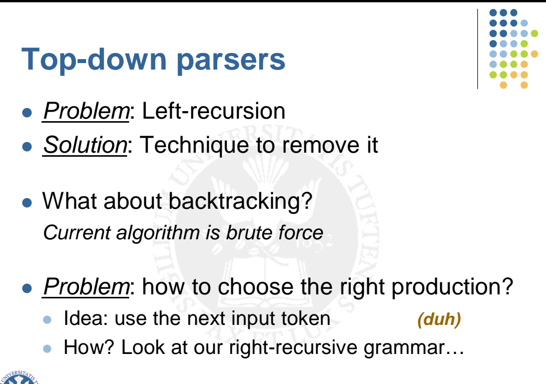

Other possible parses
Rule | Sentential form | Input string
- expr ↑ x - 2 * y
2 expr + term ↑ x - 2 * y
2 expr + term + term ↑ x - 2 * y
2 expr + term + term + term ↑ x - 2 * y
2 expr + term + term + term + term ↑ x - 2 * y
Problem: termination
Wrong choice leads to infinite expansion
(More importantly: without consuming any input!)
May not be as obvious as this
Our grammar is left recursive
Tufts University Computer Science
15

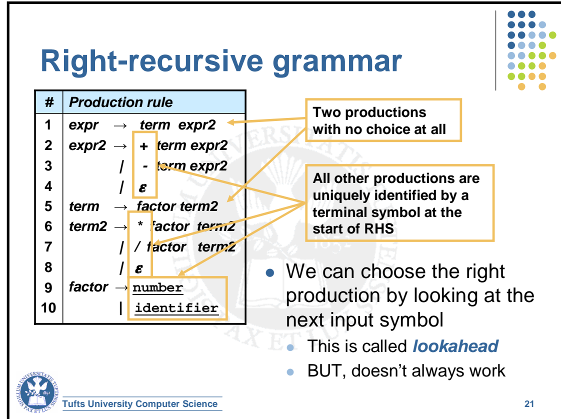

Left recursion
Formally,
A grammar is left recursive if ∃ a non-terminal A such that A →* Aα (for some set of symbols α)
Bad news:
Top-down parsers cannot handle left recursion
Good news:
We can systematically eliminate left recursion
Tufts University Computer Science
16

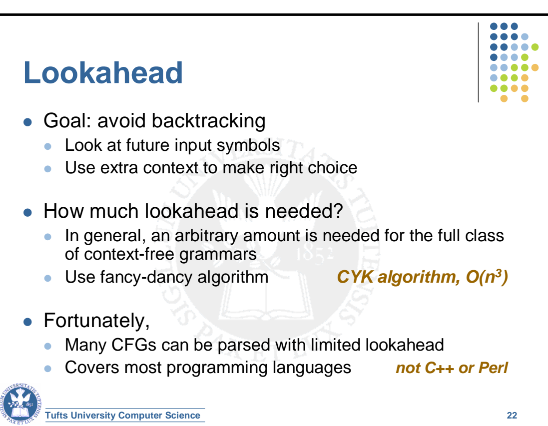

Eliminating left recursion
Consider this grammar:
| Production rule
1 foo → foo α
2 | β
Language is β followed by zero or more α
Rewrite as
| Production rule
1 foo → β bar
2 bar → α bar
3 | ε
This production gives you one β
These two productions give you zero or more α
New non-terminal
Tufts University Computer Science
17

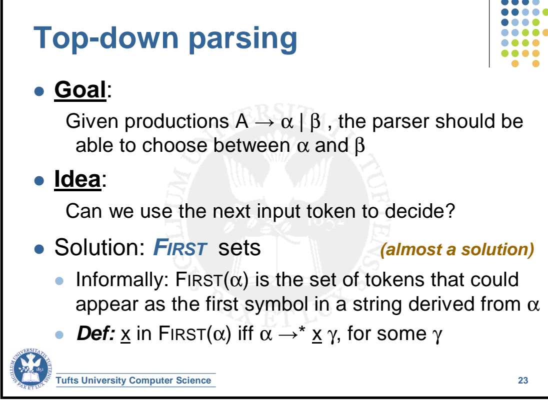

Back to expressions
Two cases of left recursion:
| Production rule
1 expr → expr + term
2 | expr - term
3 | term
| Production rule
4 term → term * factor
5 | term / factor
6 | factor
Transform as follows:
| Production rule
1 expr → term expr2
2 expr2 → + term expr2
3 | - term expr2
4 | ε
| Production rule
4 term → factor term2
5 term2 → * factor term2
6 | / factor term2
| ε
Tufts University Computer Science
18

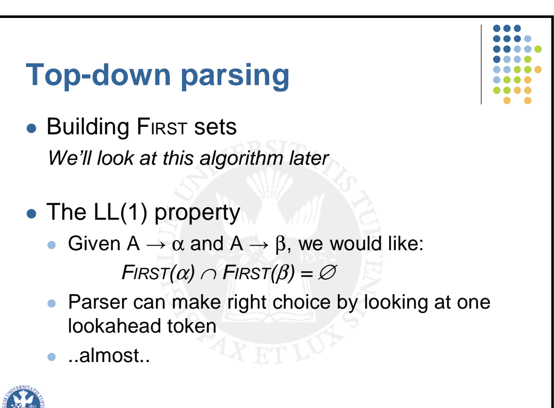

Eliminating left recursion
Resulting grammar
All right recursive
Retain original language and associativity
Not as intuitive to read
Top-down parser
Will always terminate
May still backtrack
There's a lovely algorithm to do this automatically, which we will skip
Production rule
1 expr → term expr2
2 expr2 → + term expr2
3 | - term expr2
4 | ε
5 term → factor term2
6 term2 → * factor term2
7 | / factor term2
8 | ε
9 factor → number
10 | identifier
Tufts University Computer Science
19

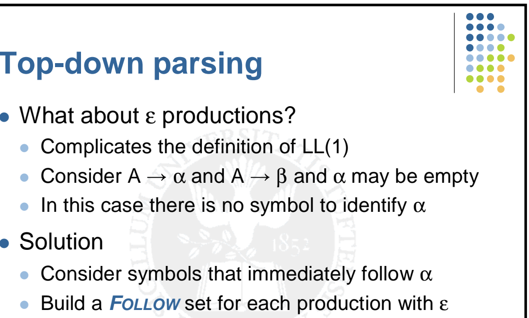

Top-down parsers
Problem: Left-recursion
Solution: Technique to remove it
What about backtracking?
Current algorithm is brute force
Problem: how to choose the right production?
Idea: use the next input token
(duh)
How? Look at our right-recursive grammar…
Tufts University Computer Science
20

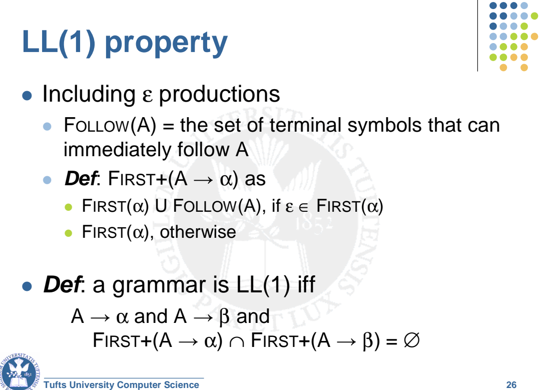

Right-recursive grammar
Production rule
1 expr → term expr2
2 expr2 → + term expr2
3 | - term expr2
4 | ε
5 term → factor term2
6 term2 → * factor term2
7 | / factor term2
8 | ε
9 factor → number
10 | identifier
Two productions with no choice at all
All other productions are uniquely identified by a terminal symbol at the start of RHS
We can choose the right production by looking at the next input symbol
This is called lookahead
BUT, doesn't always work
Tufts University Computer Science
21

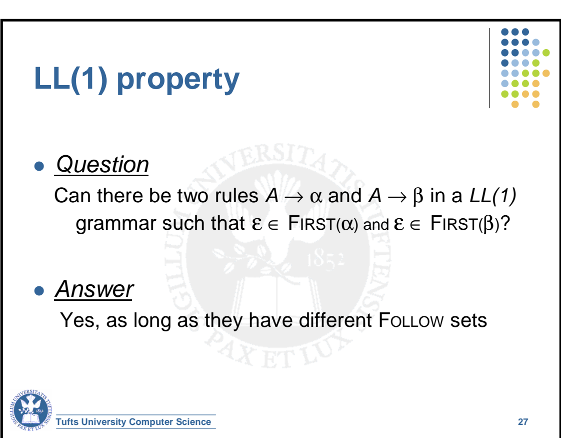

Lookahead
Goal: avoid backtracking
Look at future input symbols
Use extra context to make right choice
How much lookahead is needed?
In general, an arbitrary amount is needed for the full class of context-free grammars
Use fancy-dancy algorithm
CYK algorithm, O(n3)
Fortunately,
Many CFGs can be parsed with limited lookahead
Covers most programming languages
not C++ or Perl
Tufts University Computer Science
22

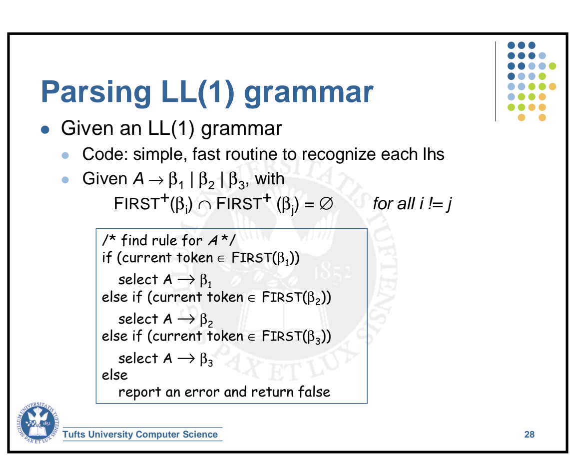

Top-down parsing
Goal:
Given productions A → α | β , the parser should be able to choose between α and β
Idea:
Can we use the next input token to decide?
Solution: FIRST sets
(almost a solution)
Informally: FIRST(α) is the set of tokens that could appear as the first symbol in a string derived from α
Def: x in FIRST(α) iff α →* x γ, for some γ
Tufts University Computer Science
23

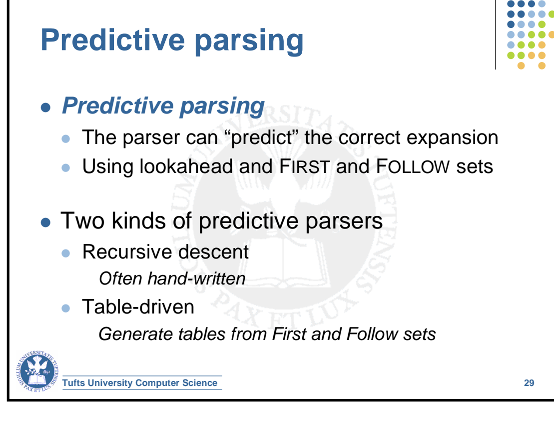

Top-down parsing
Building FIRST sets
We'll look at this algorithm later
The LL(1) property
Given A → α and A → β, we would like:
FIRST(α) ∩ FIRST(β) = ∅
Parser can make right choice by looking at one lookahead token
..almost..
Tufts University Computer Science
24

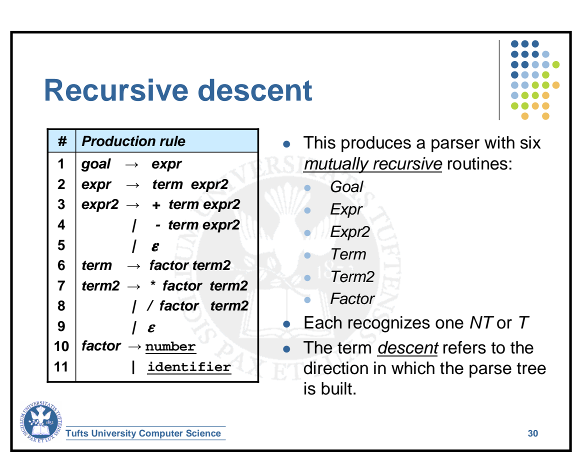

Top-down parsing
What about ε productions?
Complicates the definition of LL(1)
Consider A → α and A → β and α may be empty
In this case there is no symbol to identify α
Solution
Consider symbols that immediately follow α
Build a FOLLOW set for each production with ε
Extra condition:
FIRST(β) must be disjoint from FIRST(α) and FOLLOW(A)
Tufts University Computer Science
25

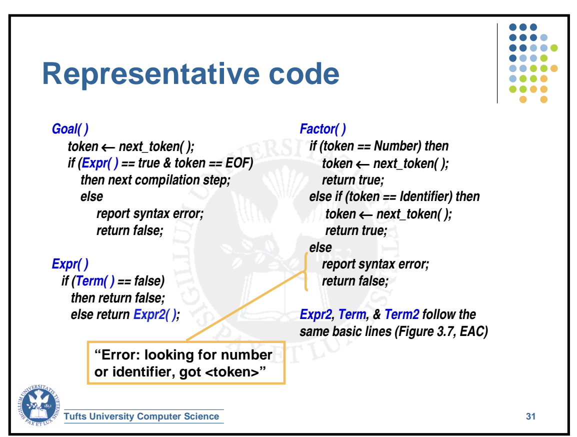

LL(1) property
Including ε productions
FOLLOW(A) = the set of terminal symbols that can immediately follow A
Def: FIRST+(A → α) as
FIRST(α) U FOLLOW(A), if ε ∈ FIRST(α)
FIRST(α), otherwise
Def: a grammar is LL(1) iff
A → α and A → β and
FIRST+(A → α) ∩ FIRST+(A → β) = ∅
Tufts University Computer Science
26

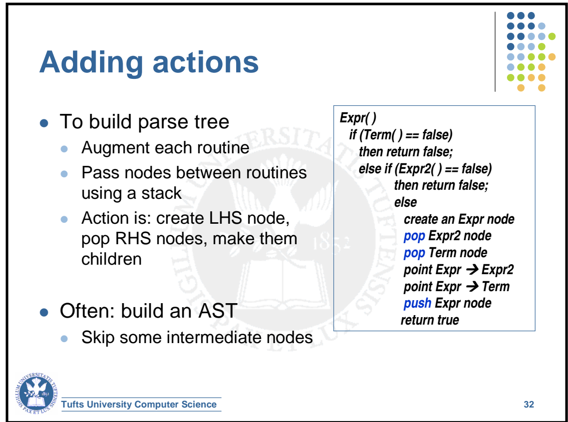

LL(1) property
Question
Can there be two rules A → α and A → β in a LL(1) grammar such that ε ∈ FIRST(α) and ε ∈ FIRST(β)?
Answer
Yes, as long as they have different FOLLOW sets
Tufts University Computer Science
27

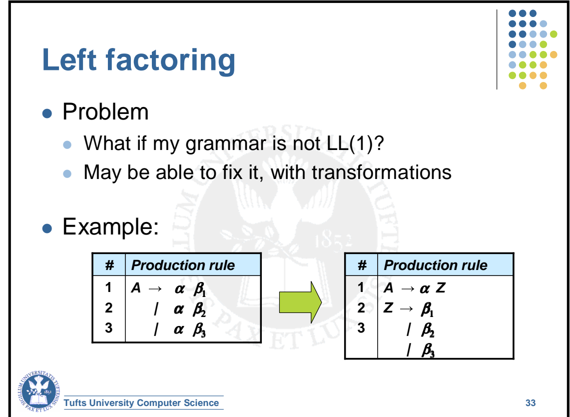

Parsing LL(1) grammar
Given an LL(1) grammar
Code: simple, fast routine to recognize each lhs
Given $A \rightarrow \beta_1 \mid \beta_2 \mid \beta_3$, with
$FIRST^+(\beta_i) \cap FIRST^+(\beta_j) = \emptyset$ for all i != j
/* find rule for A */
if (current token ∈ $FIRST(\beta_1)$)
select $A \rightarrow \beta_1$
else if (current token ∈ $FIRST(\beta_2)$)
select $A \rightarrow \beta_2$
else if (current token ∈ $FIRST(\beta_3)$)
select $A \rightarrow \beta_3$
else
report an error and return false
Tufts University Computer Science
28

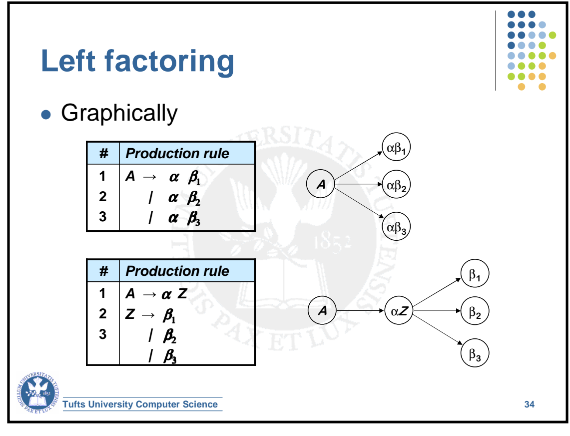

Predictive parsing
Predictive parsing
The parser can "predict" the correct expansion
Using lookahead and FIRST and FOLLOW sets
Two kinds of predictive parsers
Recursive descent
Often hand-written
Table-driven
Generate tables from First and Follow sets
Tufts University Computer Science
29

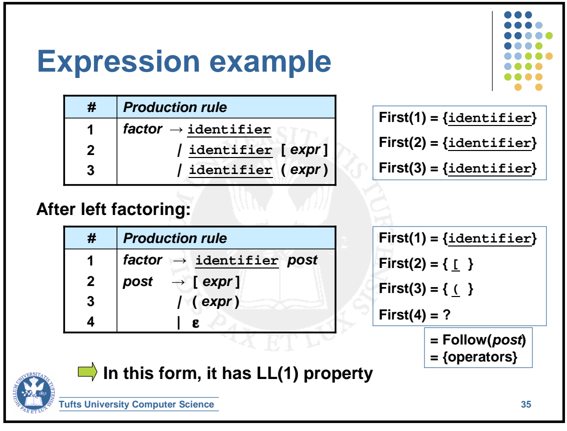

Recursive descent
Production rule
1 goal → expr
2 expr → term expr2
3 expr2 → + term expr2
4 | - term expr2
5 | ε
6 term → factor term2
7 term2 → * factor term2
8 | / factor term2
9 | ε
10 factor → number
11 | identifier
This produces a parser with six mutually recursive routines:
Goal
Expr
Expr2
Term
Term2
Factor
Each recognizes one NT or T
The term descent refers to the direction in which the parse tree is built.
Tufts University Computer Science
30

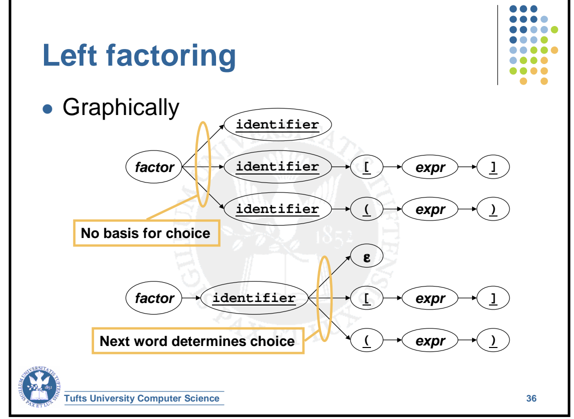

Representative code
Goal( )
token ← next_token( );
if (Expr( ) == true & token == EOF)
then next compilation step;
else
report syntax error;
return false;
Expr( )
if (Term( ) == false)
then return false;
else return Expr2( );
Factor( )
if (token == Number) then
token ← next_token( );
return true;
else if (token == Identifier) then
token ← next_token( );
return true;
else
report syntax error;
return false;
Expr2, Term, & Term2 follow the
same basic lines (Figure 3.7, EAC)
"Error: looking for number
or identifier, got <token>"
Tufts University Computer Science
31


Adding actions
To build parse tree
Augment each routine
Pass nodes between routines
using a stack
Action is: create LHS node,
pop RHS nodes, make them
children
Often: build an AST
Skip some intermediate nodes
Expr( )
if (Term( ) == false)
then return false;
else if (Expr2( ) == false)
then return false;
else
create an Expr node
pop Expr2 node
pop Term node
point Expr → Expr2
point Expr → Term
push Expr node
return true
Tufts University Computer Science
32

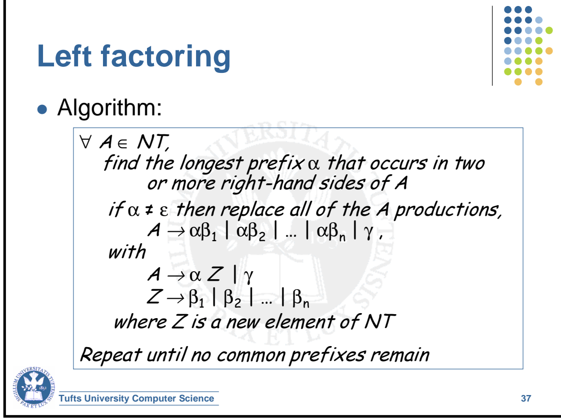

Left factoring
Problem
What if my grammar is not LL(1)?
May be able to fix it, with transformations
Example:
Production rule
1 A → α β1
2 | α β2
3 | α β3
Production rule
1 A → α Z
2 Z → β1
3 | β2
| β3
Tufts University Computer Science
33


Left factoring
Graphically
Production rule
1 A → α β1
2 | α β2
3 | α β3
A
αβ1
αβ2
αβ3
Production rule
1 A → α Z
2 Z → β1
3 | β2
| β3
A
αZ
β1
β2
β3
Tufts University Computer Science
34


Expression example
Production rule
1 factor → identifier
2 | identifier [expr]
3 | identifier (expr)
First(1) = {identifier}
First(2) = {identifier}
First(3) = {identifier}
After left factoring:
Production rule
1 factor → identifier post
2 post → [expr]
3 | (expr)
4 | ε
First(1) = {identifier}
First(2) = { [}
First(3) = { (}
First(4) = ?
= Follow(post)
= {operators}
In this form, it has LL(1) property
Tufts University Computer Science
35

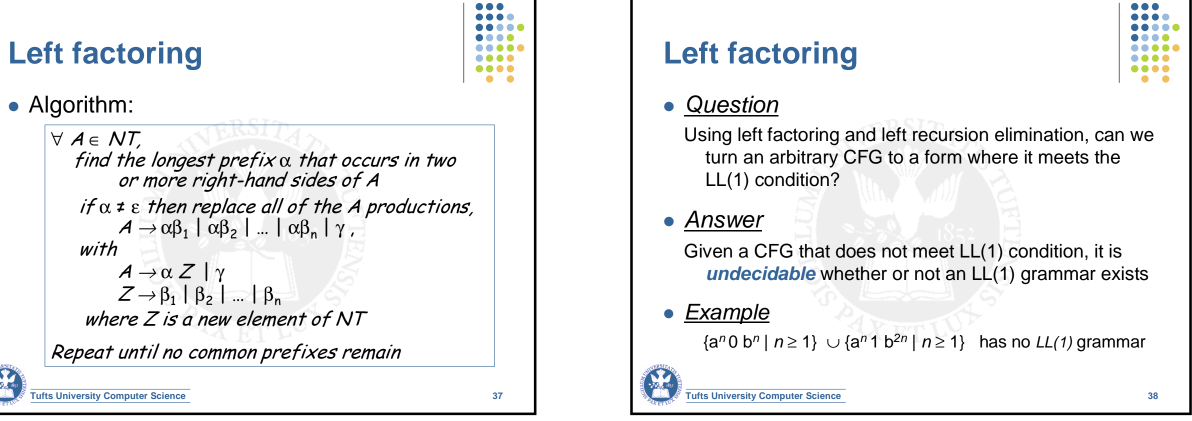

Left factoring
Graphically
factor
identifier
identifier
[
expr
]
identifier
(
expr
)
No basis for choice
factor
identifier
ε
[
expr
]
(
expr
)
Next word determines choice
Tufts University Computer Science
36

Left factoring



- Algorithm:

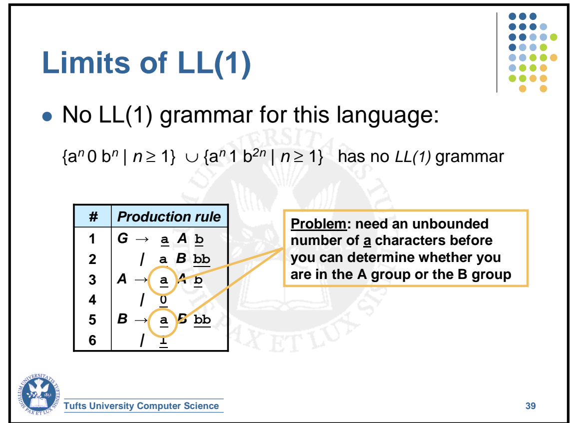
∀ A ∈ NT,
find the longest prefix α that occurs in two or more right-hand sides of A
if α ≠ ε then replace all of the A productions,
$A \rightarrow \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n \mid \gamma$,
with
$A \rightarrow \alpha Z \mid \gamma$
$Z \rightarrow \beta_1 \mid \beta_2 \mid \ldots \mid \beta_n$
where Z is a new element of NT
Repeat until no common prefixes remain



Left factoring

- *Question*
  Using left factoring and left recursion elimination, can we turn an arbitrary CFG to a form where it meets the LL(1) condition?
- *Answer*
  Given a CFG that does not meet LL(1) condition, it is ***undecidable*** whether or not an LL(1) grammar exists
- *Example*
  $\{a^n 0\, b^n \mid n \geq 1\} \cup \{a^n 1\, b^{2n} \mid n \geq 1\}$ has no *LL(1)* grammar

Limits of LL(1)

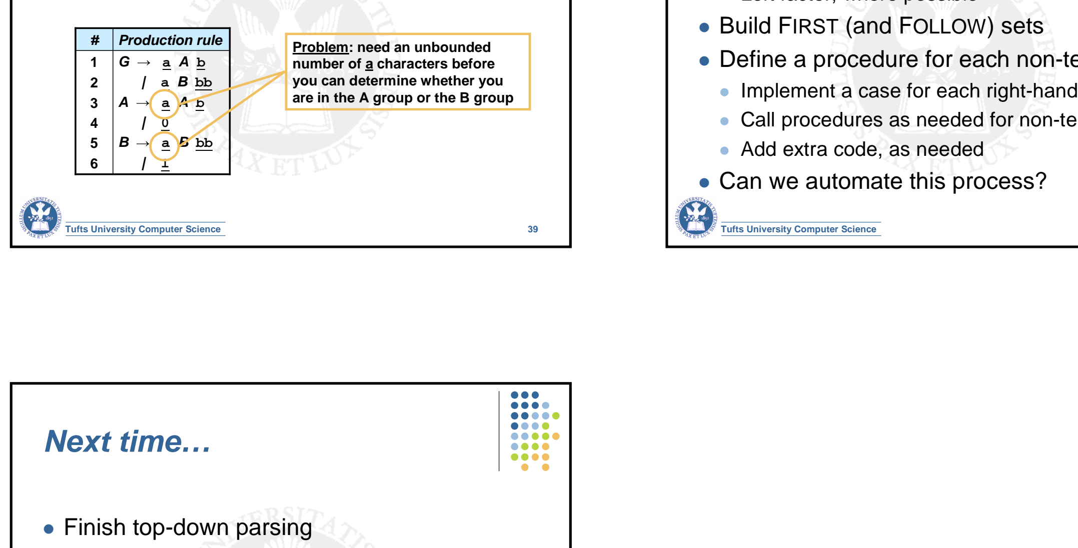


- No LL(1) grammar for this language:

$\{a^n 0\, b^n \mid n \geq 1\} \cup \{a^n 1\, b^{2n} \mid n \geq 1\}$ has no *LL(1)* grammar

#	Production rule
1	*G* → a *A* b
2	| a *B* bb
3	*A* → a *A* b
4	| 0
5	*B* → a *B* bb
6	| 1

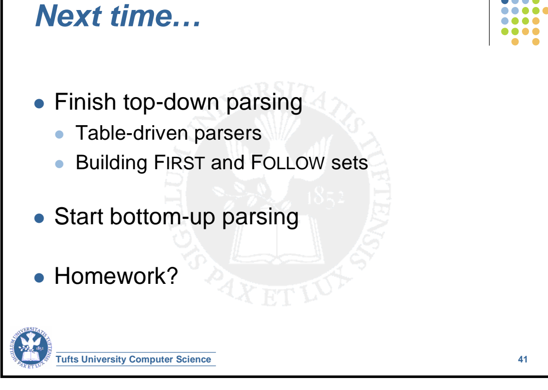
Problem: need an unbounded number of a characters before you can determine whether you are in the A group or the B group



Recursive descent parsing

- Massage grammar to have LL(1) condition
  - Remove left recursion
  - Left factor, where possible
- Build FIRST (and FOLLOW) sets
- Define a procedure for each non-terminal
  - Implement a case for each right-hand side
  - Call procedures as needed for non-terminals
  - Add extra code, as needed
- Can we automate this process?

Next time…



- Finish top-down parsing
  - Table-driven parsers
  - Building FIRST and FOLLOW sets
- Start bottom-up parsing
- Homework?