Top-Down Parsing

Parsing: Review of the Big Picture (1)

- Context-free grammars (CFGs)
  - Generation: $G \rightarrow L(G)$
  - Recognition: Given w, is $w \in L(G)$?
- Translation
  - Given $w \in L(G)$, create a (G) parse tree for w
  - Given $w \in L(G)$, create an AST for w
    - The AST is passed to the next component of our compiler

Parsing: Review of the Big Picture (2)

- Algorithms
  - CYK
  - Top-down (“recursive-descent”) for LL(1) grammars
    - How to parse, given the appropriate parse table for G
    - How to construct the parse table for G
  - Bottom-up for LALR(1) grammars
    - How to parse, given the appropriate parse table for G
    - How to construct the parse table for G

Last time

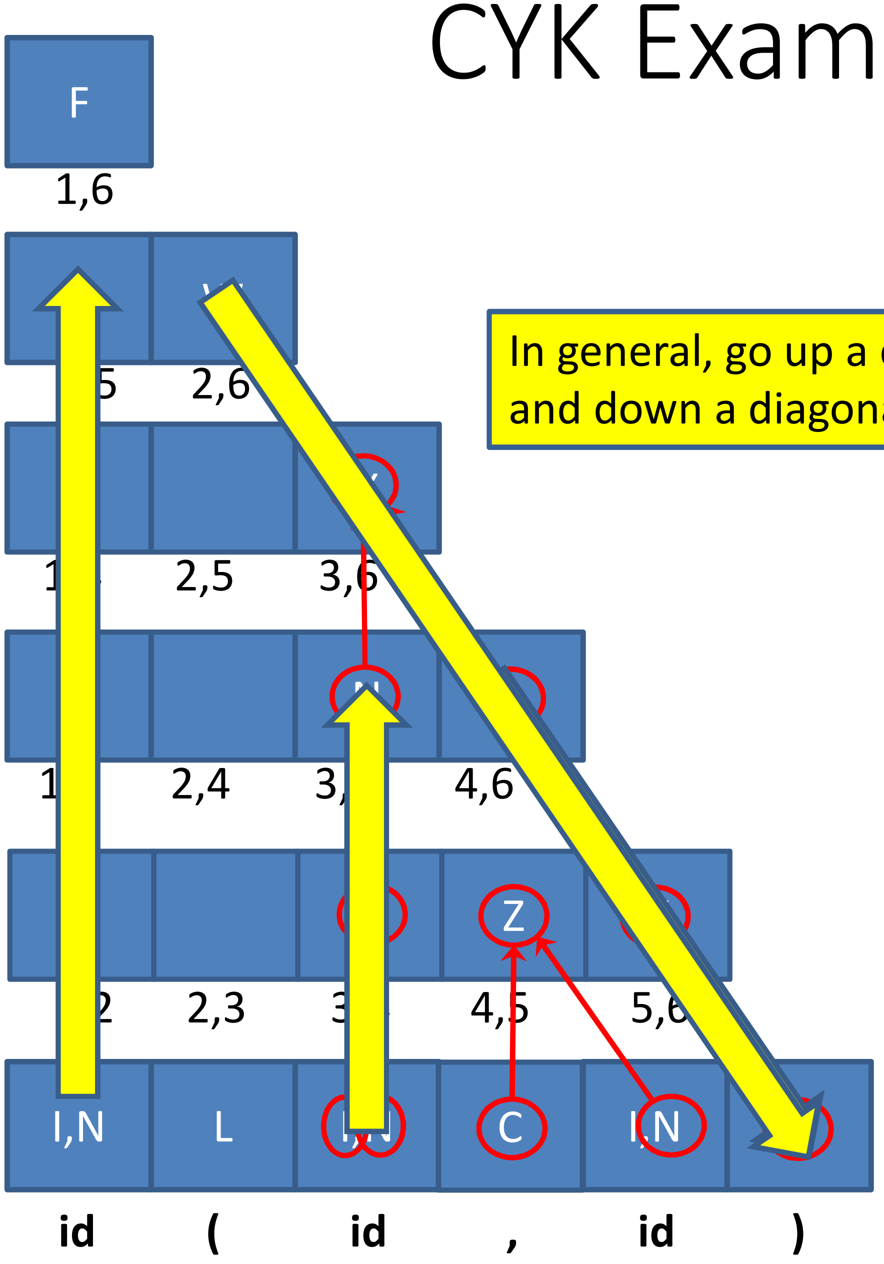
CYK

- Step 1: get a grammar in Chomsky Normal Form
- Step 2: Build all possible parse trees bottom-up
  - Start with runs of 1 terminal
  - Connect 1-terminal runs into 2-terminal runs
  - Connect 1- and 2- terminal runs into 3-terminal runs
  - Connect 1- and 3- or 2- and 2- terminal runs into 4 terminal runs
  - …
  - If we can connect the entire tree, rooted at the start symbol, we've found a valid parse

Some Interesting Properties of CYK

- Very old algorithm
  - Already well known in early 70s
- No problems with ambiguous grammars:
  - Gives a solution for *all* possible parse tree simultaneously

CYK Example



In general, go up a column and down a diagonal

F → I W

F → I Y

W → L X

X → N R

Y → L R

N → **id**

N → I Z

Z → C N

I → **id**

L → **(**

R → **)**

C → **,**

Thinking about Language Design

Balanced considerations

- Powerful enough to be useful
- Simple enough to be parsable

Syntax need not be complex for complex behaviors

- Guy Steele's "Growing a Language"

Video: https://www.youtube.com/watch?v=_ahvzDzKdB0

Text: http://www.cs.virginia.edu/~evans/cs655/readings/steele.pdf

Restricting the Grammar

By restricting our grammars we can

- Detect ambiguity
- Build linear-time, O(n) parsers

LL(1) languages

- Particularly amenable to parsing
- Parsable by <u>predictive</u> (top-down) parsers
  - Sometimes called “recursive-descent parsers”

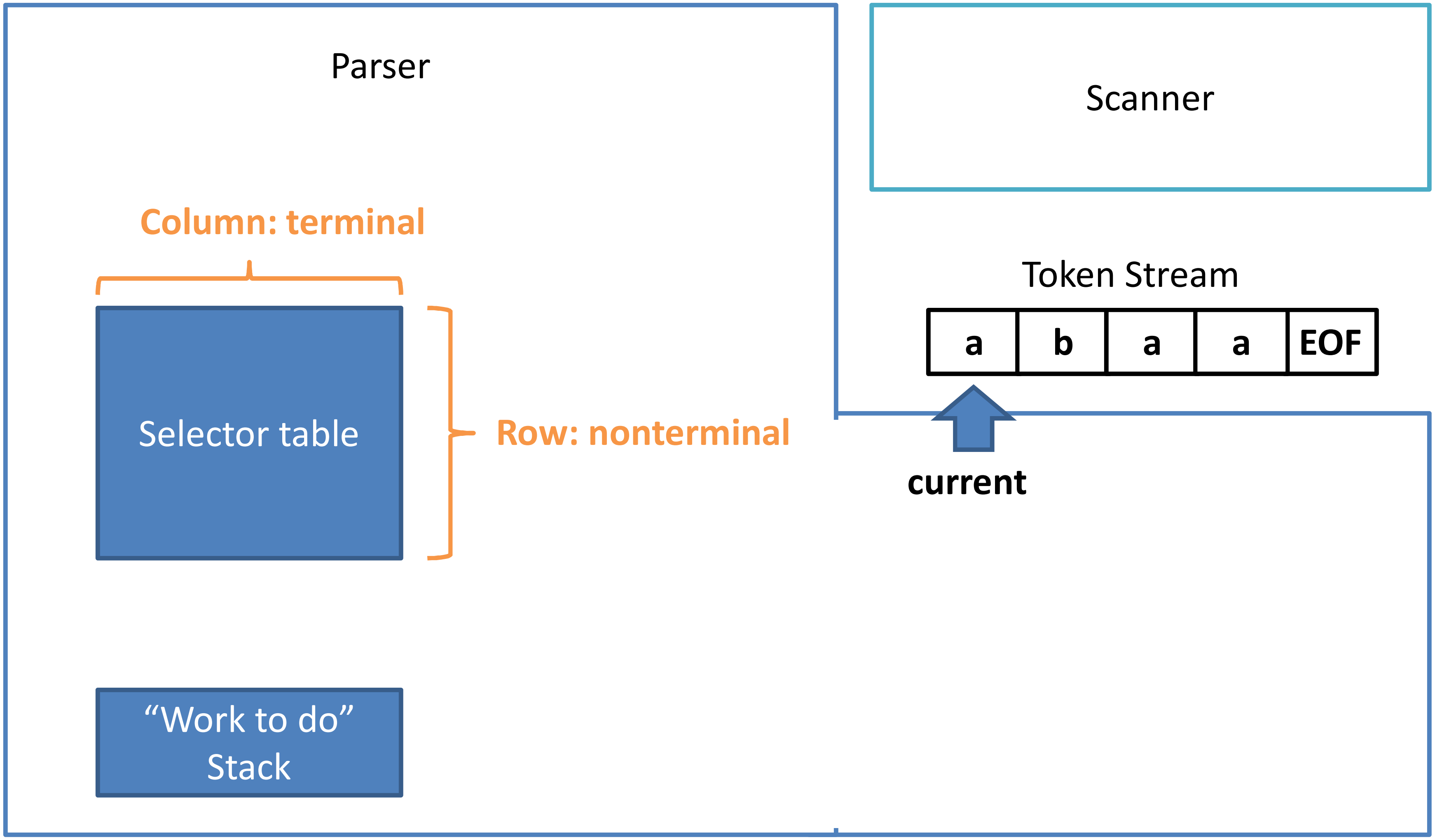
Top-Down Parsers

Start at the Start symbol

Repeatedly: “predict” what production to use

- Example: if the current token to be parsed is an id, no need to try productions that start with intLiteral
- This might seem simple, but keep in mind that a chain of productions may have to be used to get to the rule that handles, e.g., id

Predictive Parser Sketch



Parser

Column: terminal

Selector table

Row: nonterminal

“Work to do” Stack

Scanner

Token Stream

a	b	a	a	EOF

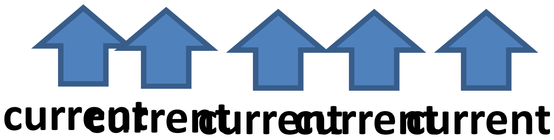
current

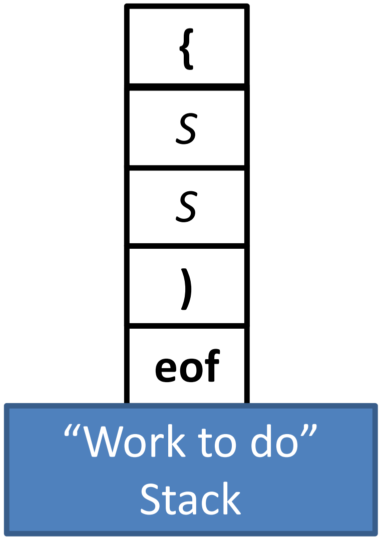
Example

$S \rightarrow$ **(** S **)** | **{** S **}** | **ε**

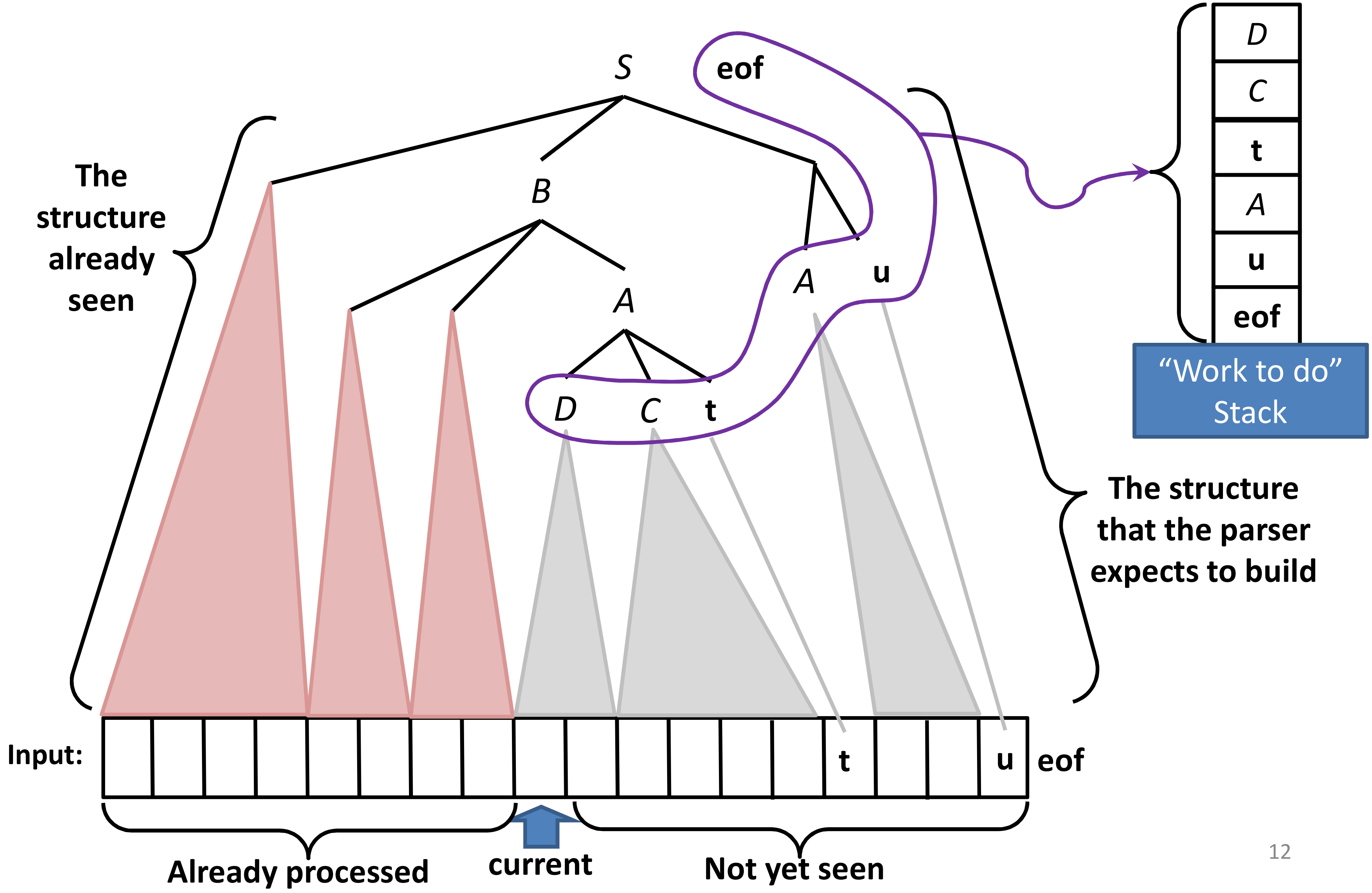
	(	**)**	**{**	**}**	**eof**
S	(S)	ε	{ S }	ε	ε

Input: **(** **{** **}** **)** **eof**





A Snapshot of a Predictive Parser



The structure already seen

The structure that the parser expects to build

"Work to do" Stack

Already processed

current

Not yet seen

Algorithm

```
stack.push(eof)
stack.push(Start non-term)
```

Initial stack is "*Start* **eof**"

```
t = scanner.getToken()
Repeat
  if stack.top is a terminal y
    match y with t
    pop y from the stack
    t = scanner.next_token()
  if stack.top is a nonterminal X
    get table[X,t]
    pop X from the stack
    push production's RHS (each symbol from Right to Left)
Until one of the following:
  stack is empty
```

accept

```
  stack.top is a terminal that does not match t
  stack.top is a non-term and parse-table entry is empty
```

reject

Example 2, bad input: You try

$S \rightarrow \mathbf{(}\ S\ \mathbf{)} \mid \mathbf{\{}\ S\ \mathbf{\}} \mid \boldsymbol{\varepsilon}$

	(	**)**	**{**	**}**	**eof**
S	(S)	ε	{ S }	ε	ε

INPUT

((} eof

This Parser Works Great!

Given a single token we always knew exactly what production it started

	(	**)**	**{**	**}**	**eof**
S	(S)	ε	{ S }	ε	ε

Two Outstanding Issues

1. How do we know if the language is LL(1)
   - Easy to imagine a grammar where a single token is not enough to select a rule

$$S \rightarrow \mathbf{(}\ S\ \mathbf{)} \mid \mathbf{\{}\ S\ \mathbf{\}} \mid \varepsilon \mid \mathbf{(\ )}$$

1. How do we build the selector table?
   - It turns out that there is one answer to both:

If our selector table has 1 production per cell, then grammar is LL(1)

LL(1) Grammar Transformations

Necessary (but not sufficient conditions) for LL(1) parsing:

- Free of left recursion
  - “No left-recursive rules”
  - Why? Need to look past the list to know when to cap it
- Left-factored
  - “No rules with a common prefix, for any nonterminal”
  - Why? We would need to look past the prefix to pick the production

Left-Recursion

- Recall that a grammar for which $X \Rightarrow^{+} X\ \alpha$ is <u>left recursive</u>
- A grammar is <u>immediately left recursive</u> if the repetition of the LHS nonterminal can happen in one step, e.g.,

$$A \rightarrow A\ \alpha \mid \beta$$

- Fortunately, it is always possible to change the grammar to remove left recursion without changing the language it recognizes

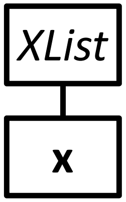
Why Left Recursion is a Problem (Blackbox View)

CFG snippet: $XList \longrightarrow XList\ \mathbf{x} \mid \mathbf{x}$

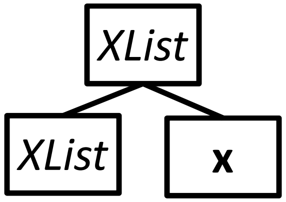
Current parse tree: *XList*

Current token: **x**

How should we grow the tree top-down?



(OR)



Correct if there are no more **x**s

Correct if there are more **x**s

We don't know which to choose without more lookahead

Why Left Recursion is a Problem (Whitebox View)

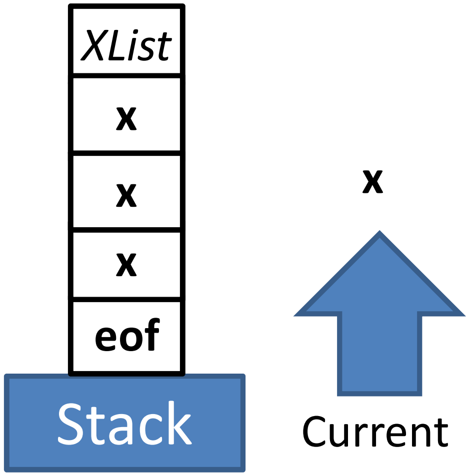
CFG snippet: *XList* ⟶ *XList* **x** | **x**

Current parse tree: *XList*

Current token: **x**

Parse table:

	x	**eof**
XList	*XList* **x**	ε



(Stack overflow)

Removing Left-Recursion

(for a single immediately left-recursive rule)

$A \rightarrow A\ \alpha \mid \beta$

$$
\begin{aligned}
A &\rightarrow \beta\ A' \\
A' &\rightarrow \alpha\ A' \\
&\mid\ \varepsilon
\end{aligned}
$$

Where β does
not begin with A

Example

A → A α | β

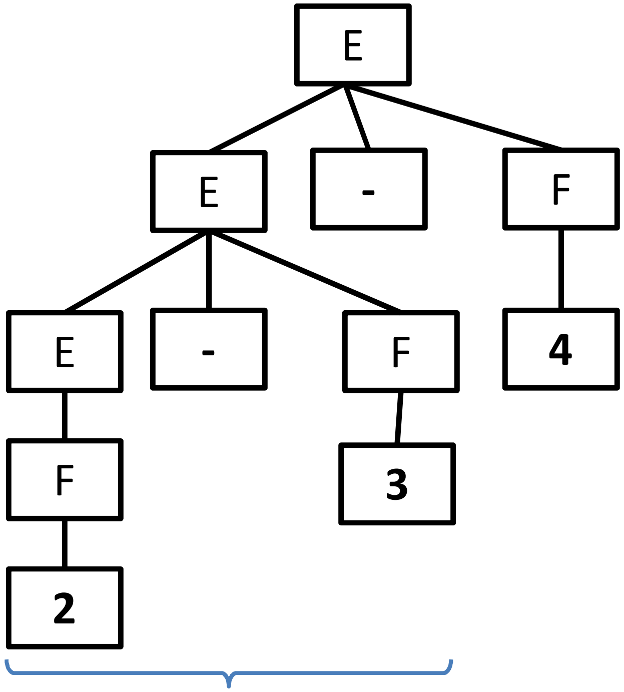
A → β *A′*
A′→ α *A′*
| ε

Exp → *Exp* **–** *Factor*
| *Factor*
Factor → **intlit** | **(** Exp **)**

Exp → *Factor Exp’*
Exp’ → - *Factor Exp’*
| ε
Factor → **intlit** | **(** Exp **)**

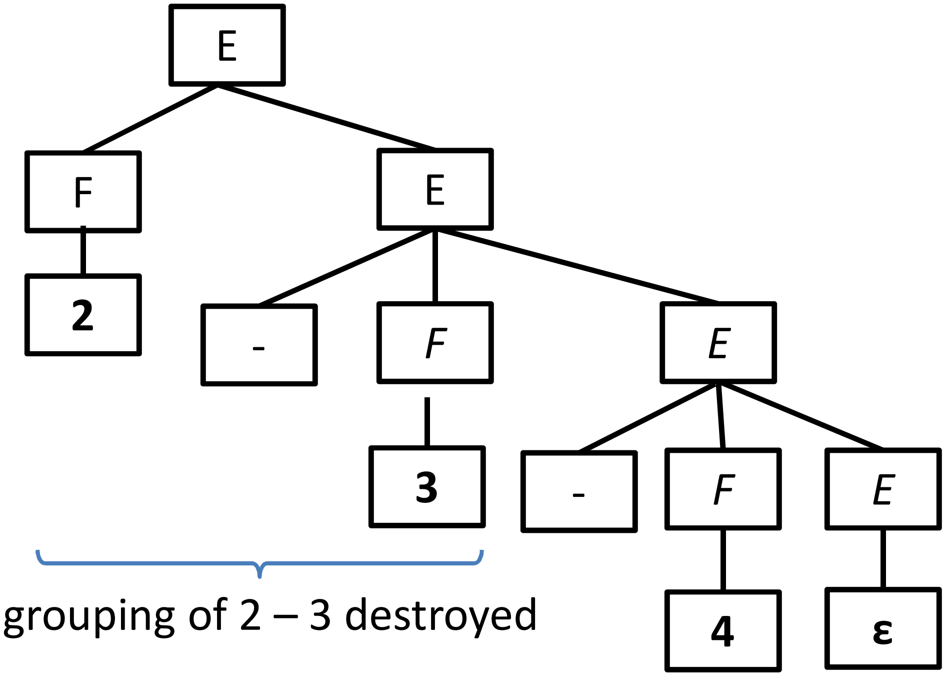
Let’s check in on the parse tree…

Exp → *Exp* **–** *Factor*
| *Factor*
Factor → **intlit** | **(** Exp **)**



2 – 3 grouped together

Exp → *Factor Exp’*
Exp’ → **-** *Factor Exp’*
| ε
Factor → **intlit** | **(** Exp **)**



grouping of 2 – 3 destroyed

☹… We’ll fix this issue later

General Rule for Removing Immediate Left-Recursion

$$A \rightarrow A\,\alpha_1 \mid A\,\alpha_2 \mid \dots \mid A\,\alpha_m \mid \beta_1 \mid \beta_2 \mid \dots \mid \beta_n$$

$$\Downarrow$$

$$A \rightarrow \beta_1\,A' \mid \beta_2\,A' \mid \dots \mid \beta_n\,A'$$
$$A' \rightarrow \alpha_1\,A' \mid \alpha_2\,A' \mid \dots \mid \alpha_m\,A' \mid \varepsilon$$

Left-Factored Grammars

If a nonterminal has two productions whose right-hand sides have a common prefix, the grammar is not left-factored, and not LL(1)

$$Exp \rightarrow (\ Exp\ )\ |\ (\ )$$

Not left-factored

Left Factoring

Given productions of the form

$$A \rightarrow \alpha\ \beta_1 \mid \alpha\ \beta_2$$

$$A \rightarrow \alpha\ A'$$

$$A' \rightarrow \beta_1 \mid \beta_2$$

Combined Example

Exp → **(** *Exp* **)** | *Exp Exp* | **( )**

Remove immediate left-recursion

Exp → **(** *Exp* **)** *Exp'* | **( )** *Exp'*
Exp' → *Exp Exp'* | ε

Left-factoring

Exp -> **(** *Exp''*
Exp'' -> *Exp* **)** *Exp'* | **)** *Exp'*
Exp' -> exp exp' | ε

Where are we at?

We’ve set ourselves up for success in building the selection table

- Two things that prevent a grammar from being LL(1) were identified and avoided
  - Left-recursive grammars
  - Non left-factored grammars
- Next time
  - Build two data structures that combine to yield a selector table:
    - FIRST sets
    - FOLLOW sets