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

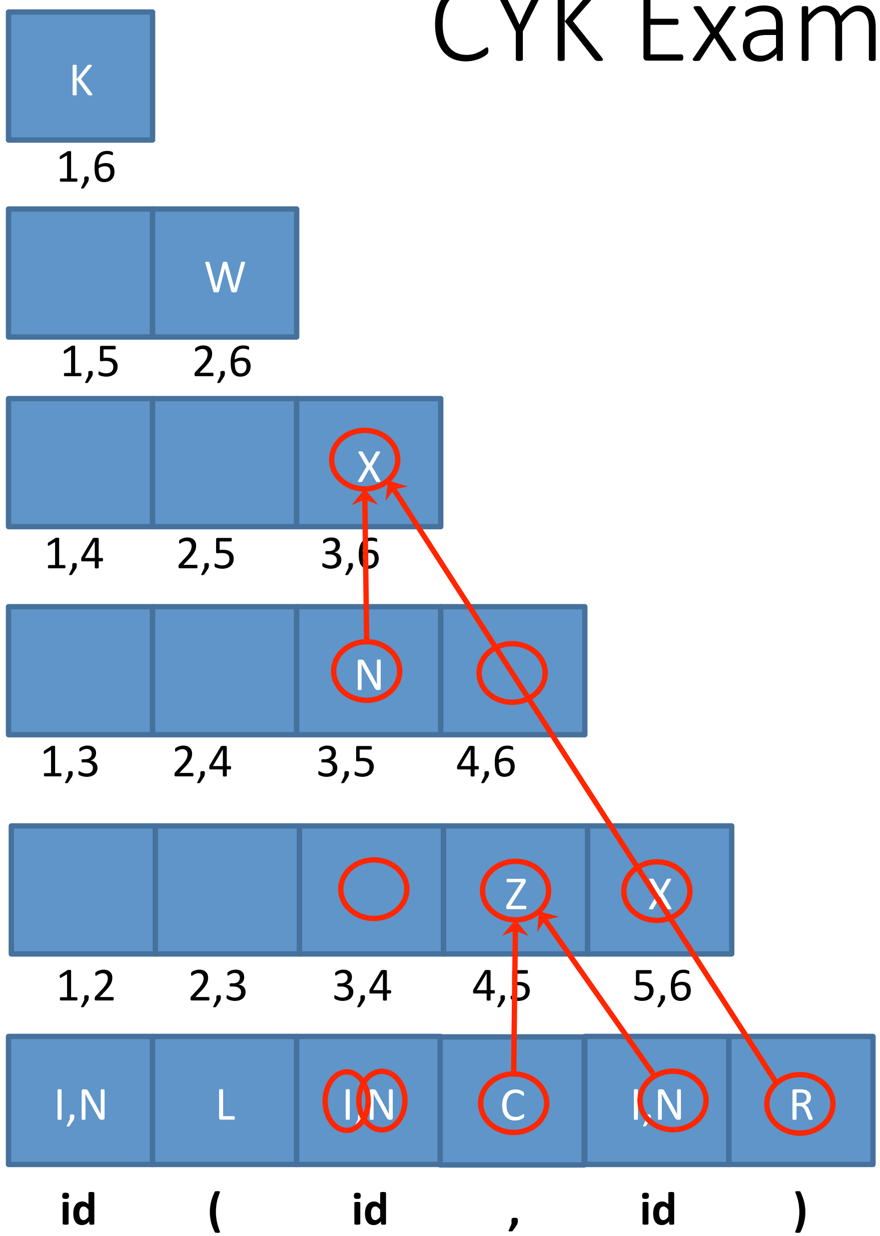
CYK

- 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



K → I W

K → 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"
  https://www.youtube.com/watch?v=_ahvzDzKdB0

Restricting the Grammar

By restricting our grammars we can

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

LL(1) languages

- Particularly amenable to parsing
- Parseable by Predictive (top-down) parsers
  - Sometimes called recursive descent

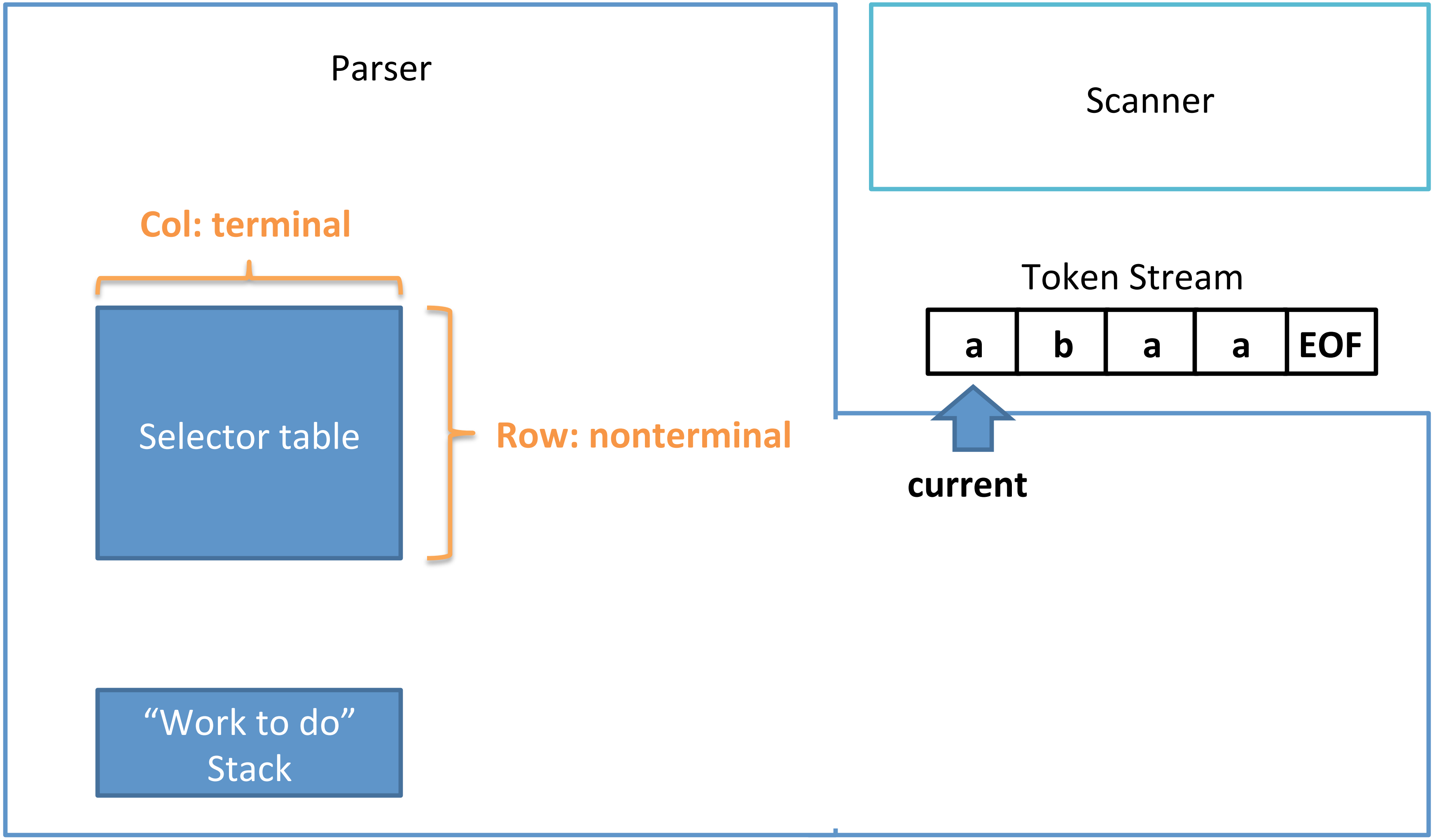
Top-Down Parsers

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**

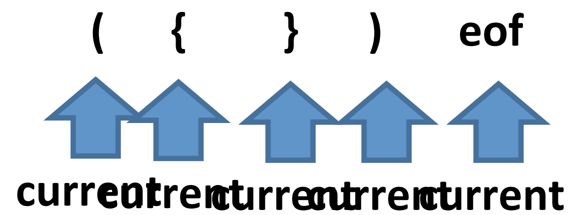
Predictive Parser Sketch



Example

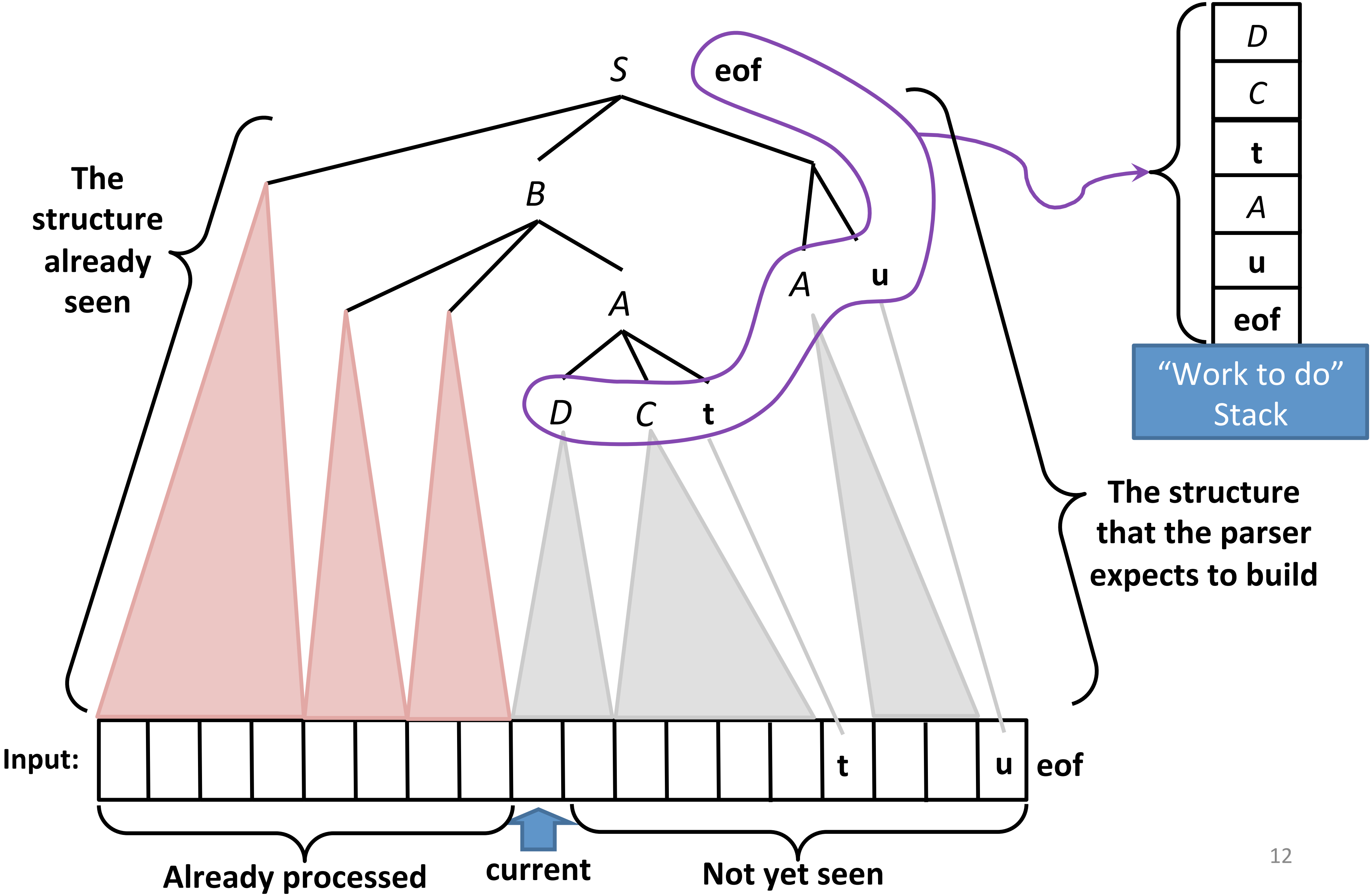
$S \rightarrow$ **(** S **)** | **{** S **}** | **ε**

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



{
S
S
)
eof
“Work to do” Stack

A Snapshot of a Predictive Parser



Algorithm

```
stack.push(eof)
stack.push(Start non-term)
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
  stack.top is a terminal that doesn't match t
  stack.top is a non-term and parse table entry is empty
```

accept

reject

Example 2, bad input: You try

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

	(	**)**	**{**	**}**	**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

Any Idea? $S \rightarrow \mathbf{(}\ S\ \mathbf{)} \mid \mathbf{\{}\ S\ \mathbf{\}} \mid \mathbf{(\ )}$

2. How do we build the selector table?

It turns out that there is one answer to both:

If 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 nonterminal loops for a production
  - Why? Need to look past list to know when to cap it
- Left factored
  - No rules with common prefix
  - Why? We’d need to look past the prefix to pick rule

Left-Recursion

Recall, a grammar such that is left recursive

A grammar is immediately left recursive if this can happen in one step:

$$A \rightarrow A\ \alpha\ |\ \beta$$

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

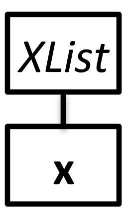
Why Left Recursion is a Problem (Blackbox View)

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

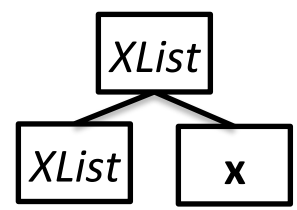
Current parse tree: $\boxed{\textit{XList}}$

Current token: **x**

How should we grow the tree top-down?



(OR)



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

Correct if there <u>are</u> more **x**s

We don’t know which without more lookahead

Why Left Recursion is a Problem (Whitebox View)

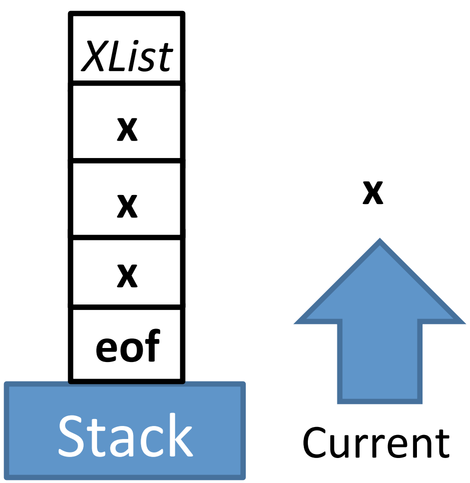
CFG snippet: *XList* ⟶ *XList* **x** | **x**

Current parse tree: $\boxed{XList}$

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

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

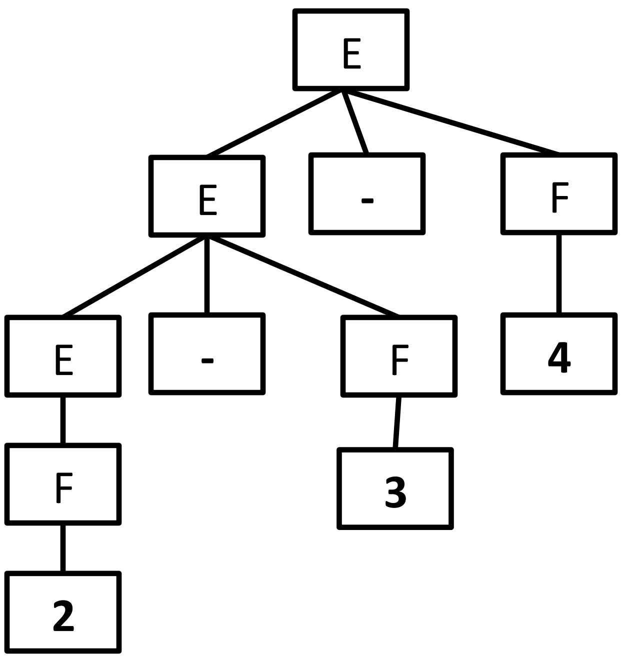
$$
\begin{aligned}
A &\rightarrow \beta\ A' \\
A' &\rightarrow \alpha\ A' \\
&\mid\ \varepsilon
\end{aligned}
$$

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

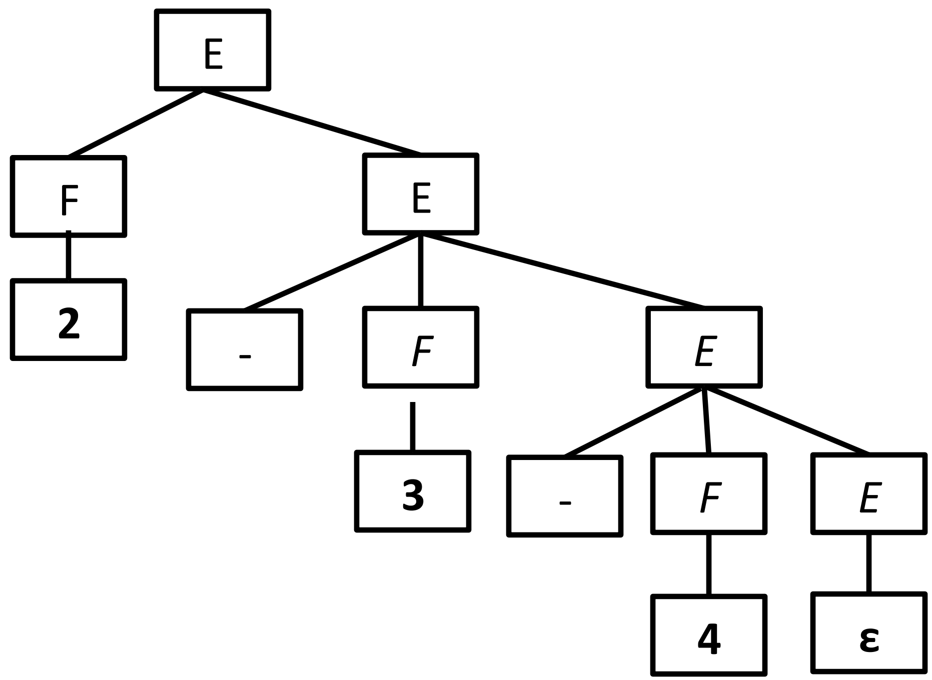
Exp → *Factor Exp'*
Exp' → **-** *Factor Exp'*
| ε
Factor → **intlit** | **(** Exp **)**

Let’s check in on the Parse Tree…

$$Exp \rightarrow Exp - Factor$$
$$\quad | \quad Factor$$
$$\text{Factor} \rightarrow \textbf{intlit} \mid (\text{ Exp })$$



$$Exp \rightarrow Factor\ Exp'$$
$$Exp' \rightarrow -\ Factor\ Exp'$$
$$\quad | \quad \varepsilon$$
$$\text{Factor} \rightarrow \textbf{intlit} \mid (\text{ Exp })$$



… We’ll fix that later

General Rule for Removing Immediate Left-Recursion

$$A \rightarrow \alpha_1 \mid \alpha_2 \mid \ldots \mid \alpha_n \mid A\ \beta_1 \mid A\ \beta_2 \mid \ldots\ A\ \beta_m$$

$$\Downarrow$$

$$A \rightarrow \alpha_1 A' \mid \alpha_2 A' \mid \ldots \mid \alpha_n A'$$
$$A' \rightarrow \beta_1 A' \mid \beta_2 A' \mid \ldots \mid \beta_m A' \mid \varepsilon$$

Left Factored Grammars

If a nonterminal has two productions whose RHS have common prefix

→ Grammar it 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
  - Not Left-Factored grammars
  - Left-recursive grammars
- Next time
  - Build two data structures that combine to yield a selector table:
    - FIRST set
    - FOLLOW set