<table>
<thead>
<tr>
<th>Introduction</th>
<th>Breaking LL Parsers</th>
<th>Eliminating Left Recursion</th>
<th>Eliminating Common Prefixes</th>
<th>FIRST/FOLLOW conflicts</th>
</tr>
</thead>
</table>

**LL Parsing**

**Dr. Mattox Beckman**

*University of Illinois at Urbana-Champaign*
*Department of Computer Science*
Objectives

The topic for this lecture is a kind of grammar that works well with recursive-descent parsing.

➤ Know how to tell if a grammar is LL.
➤ Know what parsing technique will work with an LL grammar.
➤ Know how to detect and eliminate left recursion.
➤ Know how to detect and eliminate common prefixes.
➤ Know how to detect and eliminate conflicts with first and follow sets.
What is LL(n) Parsing?

- An LL parse uses a Left-to-right scan and produces a Leftmost derivation, using n tokens of lookahead.
- A.K.A. Top-Down Parsing

Example Grammar:

```
S → + E E  
E → int  
E → * E E
```

Syntax Tree:

```
S
```

Example Input:

```
+ 2 * 3 4
```
What is \( \text{LL(n)} \) Parsing?

- An LL parse uses a \textbf{Left-to-right} scan and produces a \textbf{Leftmost} derivation, using \( n \) tokens of lookahead.
- \textbf{A.K.A. Top-Down Parsing}

\textbf{Example Grammar:}

- \( S \rightarrow + \ E \ E \)
- \( E \rightarrow \text{int} \)
- \( E \rightarrow * \ E \ E \)

\textbf{Syntax Tree:}

```
  S
   \downarrow
   +
     \_\_\_
     E
     \_\_\_
     E
```

\textbf{Example Input:}

```
+ 2 * 3 4
```
What is LL(n) Parsing?

An LL parse uses a Left-to-right scan and produces a Leftmost derivation, using n tokens of lookahead.

A.K.A. Top-Down Parsing

Example Grammar:

```
S → + E E
E → int
E → * E E
```

Syntax Tree:

```
S  E  E
+  E  2
+  E  3 4
```

Example Input:

```
+ 2  * 3 4
```
What is LL(n) Parsing?

- An LL parse uses a Left-to-right scan and produces a Leftmost derivation, using n tokens of lookahead.
- A.K.A. Top-Down Parsing

Example Grammar:

```
S → + E E
E → int
E → * E E
```

Syntax Tree:
```
S → + E E
     /   \
    +   E E
       /   \
      E E
```

Example Input:
```
+ 2 * 3 4
```
What is LL(n) Parsing?

- An LL parse uses a **Left-to-right scan** and produces a **Leftmost derivation**, using **n** tokens of lookahead.
- A.K.A. **Top-Down Parsing**

**Example Grammar:**

\[
S \rightarrow + \ E \ E \\
E \rightarrow \text{int} \\
E \rightarrow * \ E \ E
\]

**Example Input:**

+ 2 * 3 4

**Syntax Tree:**

```
     S
    / \     /
   +   E   E
  /  \ /
2   * E
  |  /
  |  E
   | /
   | E
   |/
  3
```
What is LL(n) Parsing?

- An LL parse uses a **Left-to-right scan** and produces a **Leftmost derivation**, using **n** tokens of lookahead.
- A.K.A. **Top-Down Parsing**

**Example Grammar:**

\[
S \rightarrow + \ E \ E \\
E \rightarrow \text{int} \\
E \rightarrow * \ E \ E
\]

**Example Input:**

\[
+ \ 2 \ * \ 3 \ 4
\]
How to Implement It

Interpreting a Production

- Think of a production as a function definition.
- The LHS is the function being defined.
- Terminals on RHS are commands to consume input.
- Non-terminals on RHS are subroutine calls.

- For each production, make a function of type \([\text{String}] \rightarrow (\text{Tree}, [\text{String}])\)
  - input is a list of tokens
  - output is a syntax tree and remaining tokens.
- Of course, you need to create a type to represent your tree.
Things to Notice

Key Point — Prediction

► Each function immediately checks the first token of the input string to see what to do next.

```
getE [] = undefined
getE ('*':xs) =
    let e1,r1 = getE xs
    e2,r2 = getE r1
    in (ETimes e1 e2, r2)
getE .... -- other code follows
```
Left Recursion

Left Recursion is Bad

- A rule like $E \rightarrow E \; + \; E$ would cause an infinite loop.

```haskell
getE xx =
    let e1,r1 = getE xx
        ('+':r2) = r1
        e2,r3 = getE r2
    in (EPlus e1 e2, r3)
```
Rules with Common Prefixes

Common Prefixes are Bad

- A pair of rules rule like \[ E \rightarrow \quad - E \quad \mid \quad - E E \] would confuse the function.

Which version of the rule should be used?

1. `getE ('-':xs) = ... -- unary rule`
2. `getE ('-':xs) = ... -- binary rule`

- NB: Common prefixes must be for the same non-terminal. E.g., \( E \rightarrow x A \) and \( S \rightarrow x B \) do not count as common prefixes.
The Idea

Consider deriving $i++++$ from the following grammar:

$E \rightarrow E +$  "We can have as many +s as we want at the end of the sentence."

$E \rightarrow i$  "The first word must be an i"
More complicated example

Consider the following grammar. What does it mean?

\[ B \rightarrow Bxy \mid Bz \mid q \mid r \]

- At the end can come any combination of \( x \), \( y \) or \( z \)
- At the beginning can come \( q \) or \( r \)
Eliminating the Left Recursion

We can rewrite these grammars

\[ E \rightarrow E + \mid i \]
\[ B \rightarrow Bxy \mid Bz \mid q \mid r \]

using the following transformation:

- Productions of the form \( S \rightarrow \beta \) become \( S \rightarrow \beta S' \).
- Productions of the form \( S \rightarrow S\alpha \) become \( S' \rightarrow \alpha S' \).
- Add \( S' \rightarrow \epsilon \).

Result:

\[ E \rightarrow iE' \]
\[ E' \rightarrow +E' \mid \epsilon \]
\[ B \rightarrow qB' \mid rB' \]
\[ B' \rightarrow xyB' \mid zB' \mid \epsilon \]
Mutual Recursions!

Things are slightly more complicated if we have mutual recursions.

\[
A \rightarrow Aa \mid Bb \mid Cc \mid q \\
B \rightarrow Ax \mid By \mid Cz \mid rA \\
C \rightarrow Ai \mid Bj \mid Ck \mid sB
\]

How to do it:

- Take the first symbol (A) and eliminate immediate left recursion.
- Take the second symbol (B), and substitute left recursions to A. Then eliminate immediate left recursion in B.
- Take the third symbol (C) and substitute left recursions to A and B. Then eliminate immediate left recursion in C.
Left Recursion Example

Here is a more complex left recursion.

\[
A \rightarrow Aa \mid Bb \mid Cc \mid q \\
B \rightarrow Ax \mid By \mid Cz \mid rA \\
C \rightarrow Ai \mid Bj \mid Ck \mid sB
\]

First we eliminate the left recursion from \( A \).

\[
A \rightarrow Aa \mid Bb \mid Cc \mid q
\]

becomes

\[
A \rightarrow BbA' \mid CcA' \mid qA' \\
A' \rightarrow aA' \mid \epsilon
\]
Left Recursion Example, 2

We substituting in the new definition of $A$, and now we will work on the $B$ productions.

$A \rightarrow BbA' | CcA' | qA'$
$A' \rightarrow aA' | \epsilon$

$B \rightarrow Ax | By | Cz | rA$

$C \rightarrow Ai | Bj | Ck | sB$

First, we eliminate the “backward” recursion from $B$ to $A$.

$B \rightarrow Ax$ becomes

$B \rightarrow BbA'x | CcA'x | qA'x$
Left Recursion Example, 3

\[
A \rightarrow BbA' \mid CcA' \mid qA' \\
A' \rightarrow aA' \mid \epsilon \\
B \rightarrow BbA'x \mid CcA'x \mid qA'x \mid By \mid Cz \mid rA \\
C \rightarrow Ai \mid Bj \mid Ck \mid sB \\
\]

Now we can eliminate the simple left recursion in \( B \), to get

\[
B \rightarrow CcA'xB' \mid qA'xB' \mid CzB' \mid rAB' \\
B' \rightarrow bA'xB' \mid yB' \mid \epsilon 
\]
Left Recursion Example, 4

\[
\begin{align*}
A & \rightarrow BbA' \mid CcA' \mid qA' \\
A' & \rightarrow aA' \mid \epsilon \\
B & \rightarrow CcA'xB' \mid qA'xB' \mid CzB' \mid rAB' \\
B' & \rightarrow bA'xB' \mid yB' \mid \epsilon \\
C & \rightarrow Ai \mid Bj \mid Ck \mid sB
\end{align*}
\]

Now production C: first, replace left recursive calls to A...

\[
C \rightarrow \text{B} \ bA'i \mid CcA'i \mid qA'i \mid \text{B} \ j \mid Ck \mid sB
\]

Next, replace left recursive calls to B (this gets messy)...

\[
C \rightarrow \text{CcA'xB'} \ \text{bA'i} \mid \text{qA'xB'} \ \text{bA'i} \mid \text{CzB'} \ \text{bA'i} \mid \text{rAB'} \ \text{bA'i} \\
\text{CcA'xB'} \ j \mid \text{qA'xB'} \ j \mid \text{CzB'} \ j \mid \text{rAB'} \ j \\
\text{CcA'}i \mid \text{qA'}i \mid \text{Ck} \mid \text{sB}
\]
**Left Recursion Example, 5**

Reorganizing C, we have

\[
C \rightarrow \quad qA'xB'bA'i | rAB'bA'i | qA'xB'j | rAB'j | qA'i | sB \\
\quad CcA'xB'bA'i | CzB'bA'i | CcA'xB'j | CzB'j | CcA'i | Ck \\
\]

Eliminating left recursion gives us

\[
C \rightarrow \quad qA'xB'bA'iC' | rAB'bA'iC' | qA'xB'jC' \\
\quad | rAB'jC' | qA'iC' | sBC' \\
C' \rightarrow \quad cA'xB'bA'iC' | zB'bA'iC' | cA'xB'jC' \\
\quad | zB'jC' | cA'iC' | kC' | \epsilon
\]
The result...

Our final grammar is now

\[ A \to BbA' \mid CcA' \mid qA' \]
\[ A' \to aA' \mid \epsilon \]
\[ B \to CcA'xB' \mid qA'xB' \mid CzB' \mid rAB' \]
\[ B' \to bA'xB' \mid yB' \mid \epsilon \]
\[ C \to qA'xB'bA'iC' \mid rAB'bA'iC' \mid qA'xB'jC' \]
\[ \mid rAB'jC' \mid qA'iC' \mid sBC' \]
\[ C' \to cA'xB'bA'iC' \mid zB'bA'iC' \mid cA'xB'jC' \]
\[ \mid zB'jC' \mid cA'iC' \mid kC' \mid \epsilon \]

Beautiful, isn’t it? I wonder why we don’t do this more often?

▶ Disclaimer: if there is a cycle (\( A \to^+ A \)) or an epsilon production (\( A \to \epsilon \)) then this technique is not guaranteed to work.
Common Prefix

This grammar has common prefixes.

\[ A \to xyB \mid CyC \mid q \]
\[ B \to zC \mid zx \mid w \]
\[ C \to y \mid x \]

To check for common prefixes, take a non-terminal and compare the First sets of each production.

Production | FirstSet
---|---
\( A \to xyB \) | \( \{x\} \)
\( A \to CyC \) | \( \{x, y\} \)
\( A \to q \) | \( \{q\} \)

If we are viewing an \( A \), we will want to look at the next token to see which \( A \) production to use. If that token is \( x \), then which production do we use?
Left Factoring

If $A \rightarrow \alpha \beta_1 \mid \alpha \beta_2 \mid \gamma$ we can rewrite it as

$$A \rightarrow \alpha A' \mid \gamma$$

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

So, in our example:

$A \rightarrow xyB \mid CyC \mid q$ becomes $A \rightarrow xA' \mid q \mid yyC$

$B \rightarrow zC \mid zx \mid w$ becomes $A' \rightarrow yB \mid yC$

$C \rightarrow y \mid x$ becomes $B \rightarrow zB' \mid w$

$B' \rightarrow C \mid x$

$C \rightarrow y \mid x$

Sometimes you’ll need to do this more than once. Note that this process can destroy the meaning of the nonterminals.
Epsilon Productions

- Epsilon productions have to be handled with care.

\[
\begin{align*}
A & \rightarrow \ Bc \\
& \quad | \ x \\
B & \rightarrow \ c \\
& \quad | \ \epsilon
\end{align*}
\]

Is this LL?
Epsilon Productions

\[ A \rightarrow Bc \]
\[ \text{or } A \rightarrow x \]
\[ B \rightarrow c \]
\[ \text{or } B \rightarrow \epsilon \]

- \( \text{FOLLOW}(B) = \{c\} \), and \( \text{FIRST}(B) = \{c\} \), so we have a conflict when trying to parse \( B \).
- We can substitute the \( B \) rule into the \( A \) rule to fix this...
- Be sure to check if you have introduced a common prefix though!

\[ A \rightarrow cc \]
\[ \text{or } A \rightarrow c \]
\[ \text{or } A \rightarrow x \]
\[ \Rightarrow \]

\[ A \rightarrow cA' \]
\[ \text{or } A \rightarrow x \]
\[ A' \rightarrow c \]
\[ \text{or } A' \rightarrow \epsilon \]